2.20 FLIGHT POSTURE

The purpose of the Flight Posture Decision (FPD) Functional Element (FE) is to model the flight leader decision process of choosing the flight posture that will best satisfy the short-range and long-range objectives of the flight, given the current situation.

Figure 2.20-1 shows the hierarchy of decisions made by Brawler decision-makers. The flight posture decision occurs at the highest level and determines the general course of action. It is made on the basis of broad assessments of the situation, such as force ratios and engagement geometry, and also on the basis of user-supplied priorities. At the next level the flight leader determines the tactics that should be used to implement the flight posture. As a result of this decision, a specific communication is sent to other members of the flight informing them of the tactics. The effect of the message is to influence the values that other pilots use to score the alternative actions they consider. For instance, an order to attack a certain aircraft results in the subordinate perceiving that hostile as being more valuable; offensive pilot postures are weighted favorably. Below the flight tactics decision are decisions made by each individual pilot that determine his posture and his weapon and maneuver decisions in light of the ordered tactic. The primary effect of high-level decisions is to control the lower level decisions by modifying their evaluation functions and by determining which lower level alternative actions will be considered. High-level decisions can be made on the basis of more aggregated representations of the external world, resulting in considerable computational efficiency.

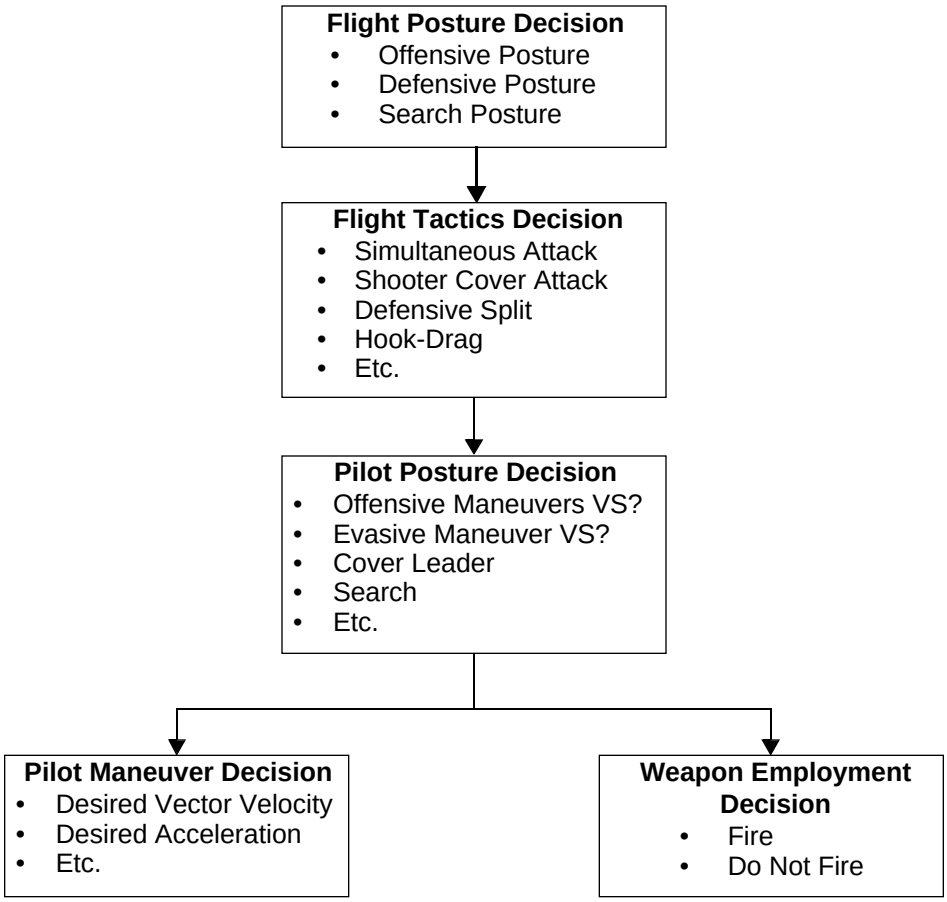

FIGURE 2.20-1. Brawler Decision Hierarchy.

There are currently eight alternative flight postures from which the FPD maker may choose. There are also flight postures associated with SAM sites and GCI/AWACS entities, which have three available postures.

To choose between the courses of action when the value-driven system is operative, predictions are made of the overall hostile value killed, *kill*, and the overall friendly value lost, *risk*. A prediction of the ability to continue the mission after engagement is made. This is done in a heuristic manner by making general judgments regarding the importance of the predicted long-term outcome of the engagement versus the importance of the expected short-term outcome of the engagement. In the case of the attack-immediate posture, for instance, the short- and long-term phases are given equal weight, while in the case of the disengage posture the long-term phase is given zero weight.

Implementation of a flight posture involves two things: setting of offensive, defensive, and mission importance multipliers that influence the value functions of other decisions lower in the hierarchy, and limiting the flight tactics that may be considered. For example, the close-from-long-range posture sets the offensive multiplier at a value of 1.0 (*agg fac*), the defensive multiplier at 0.5, and the mission multiplier at 0.1, while the disengage posture sets these values at, respectively, the maximum of one-fourth of *aggfac* or 0.5, 1.0, and 0.5. Any limiting of the flight tactics that may be considered is user controlled and causes only those flight tactics consistent with the objectives of the selected flight posture to be considered.

2.20.1 Functional Element Design Requirements

The FPD function will simulate the selection of a flight posture by a flight leader based upon information about his flight and other known friendly and hostile flights. This selection will be simulated by a process of alternative projection, and evaluation which will result in appropriate simulated action by the decision-maker. The function will be executed repeatedly for each pilot in the scenario so that a postures affecting offensive and defensive actions can vary in response to changing tactics and engagement factors over the course of simulation execution.

a. Brawler will simulate flight leader selection of flight posture from among at least eight (8) possible alternatives. The alternatives will be:

    1. **Mission**: This posture results in flight performing routine activities such as flying towards a route point or orbiting a CAP station.

    2. **Escort**: Selected when the flight is escorting a group of bombers. A value-driven decision weighing the risk to the penetrators will be made when deciding whether to engage interceptors.

    3. **GCI Mission**: Allows the flight to engage in drag tactics specified by a GCI or AWACS controller.

    4. **Attack Immediate**: The flight attacks hostiles in the within-visual-range (WVR) arena.

5. **Evade then Reengage**: The flight evades the hostiles but does not disengage; the intent is to recover after being "jumped" and then continue the engagement.

6. **Disengage**: The flight attempts to disengage safely from hostile forces.

7. **Close from Long Range**: Used to deal with the maneuvers prior to an attack. It also includes the attack phase when medium- and long-range active missiles are used.

8. **Follow GCI**: A restricted subset of 3 above; the flight is forced to follow the GCI instructions without other options.

9. A ninth flight posture **Return to Base**, is present in a rudimentary form but is not fully implemented.

b. There will also be flight postures associated with SAM sites and GCI/AWACS entities which will be:

1. **Informational**: Information concerning the location and velocity of hostiles detected by the GCI radar is broadcast to friendly flights.

2. **Intercept**: The GCI controller provides steering to the controlled flight to achieve a lag intercept of a hostile group.

3. **Drag**: The GCI or AWACS controller vectors a group of friendlies in front of a hostile flight, attempting to drag the hostiles into an attack by a second group of friendlies.

c. Brawler will simulate evaluation of each alternative posture through a process of scoring expected outcomes so that selection can be made from among ranked alterations.

d. Brawler will simulate selection of the best alternative by placing the highest scoring alternative in the pilot mental model of the flight leader.

e. Brawler will simulate actions related to the selected flight posture by simulating transmission of radio messages to the flight or other appropriate actions by the flight leader.

These requirements will be satisfied by the combined implementation of the design elements described in the following section. They were inferred from descriptions of how the existing model currently performs the FPD function.

2.20.2 Functional Element Design Approach

This section contains a description of the design approach that will implement the design requirements outlined in the previous section.

To choose between the alternative postures, predictions are made of the overall hostile value expected to be killed, denoted below by the variable *kill*, and the overall friendly value expected to be lost, denoted below by the variable *risk*, if the candidate posture is

followed. A prediction of the ability to continue the mission after engagement is also made. This is done by making general judgments regarding the importance of the predicted long-term outcome of the engagement versus the importance of the expected short-term outcome of the engagement. In the case of the attack-immediate posture, for instance, the short- and long-term phases are given equal weight, while in the case of the disengage posture the long-term phase is given zero weight.

Other general parameters characteristic of each posture specify the short-term importance of offensive and defensive factors. In the attack-immediate posture these are given full weight versus the long term considerations. In the disengage posture, because the pilots will be trying to leave the arena, short-term factors are of less importance (attack opportunities may be bypassed if trying to disengage) so each factor is given a weight of only 0.3 on a zero-to-one scale. The evade-with-intent-to-reengage posture is an intermediate case and the weights used here are 0.5.

Selection of the flight posture alternative is performed by the following procedure.

1. Generate a candidate flight posture alternative.
2. Project the candidate alternative.
3. Evaluate the projected candidate alternative.
4. Select the highest scoring candidate.
5. Schedule actions associated with the chosen alternative.

Then subroutine *aproj4* projects the outcome of that alternative over the next few seconds. Subroutine *aeval4* then evaluates or scores the outcome. The resultant value, *altvly*, is weighted with any user-specified production rule bias resulting in a net value *altvlx*. This is saved on a list and compared with the value of other alternatives. This process repeats for each candidate alternative to be considered. The best scoring alternative is selected from the list as the desired action. Unneeded alternative descriptions are deleted and the one picked is placed in the *althld* array in the /*althld*/ common block and the *cactn* array in the /*mind2*/ common block.

Design Element 20-1: Generate a Candidate Flight Posture Alternative

Subroutine *aslct4* generates one of the eight candidate flight posture alternatives each time it is called. An initialization call positions the selection algorithm at the beginning of the list of flight posture alternatives. Each successive call to the alternative generation routine *aslct4* returns the next alternative in order until the list is exhausted, at which point the *more* flag is set to *false* to indicate that there are no more alternatives. Generation of an alternative simply consists of specifying which of the available postures the alternative is. For each alternative generated, projection and evaluation are performed.

Design Element 20-2: Project the Candidate Alternative

Subroutine *aproj4* predicts the results of the candidate flight posture alternative by projecting the alternative forward in time by a few seconds. The primary results of the prediction are the anticipated value of hostiles killed (*kill*) and the anticipated value of friendlies lost (*risk*).

These factors are used along with the situational assessment to estimate the overall *risk* and *kill* values for the posture being considered. The formulas used are:

$$risk = w_a(riska) + w_s(risks)(f_d)$$

$$kill = w_a(killa) + w_s(kills)(f_o)$$

where *killa* is the long-term expected hostile kills and *riska* is the long-term expected friendly losses (both expressed in value units), and *kills* and *risks* are corresponding numbers based on a short-term situational assessment. w_a and w_s are weights ($w_a+w_s=1$) giving the relative importance of short- and long-term considerations. f_o and f_d are parameters giving the importance of offensive and defensive factors in the short-term phase of an engagement. The weights w_a and w_s are functions of the estimated importance of the long-term phase, W_a, and the expected overall destruction rates for the short- and long-term phases:

$$w_a = k(W_a)(killa^2 + riska^2)$$

$$w_s = k(1-W_a)[(kills)(f_o)^2 + (risks)(f_d)^2]$$

where k is chosen to normalize w_a+w_s. The rationale for avoiding the simpler assignment $w_a=W_a$ is that a high kill rate (overall) in the early situational phase leaves little of importance for the long-term phase. The terms situational and *a priori* are used to refer to the short- and long-term phases of the engagement since, at this aggregated level, air combat can be viewed as consisting of a short-term phase where the entry conditions for the engagement play an important role (situational), followed by a later phase which, if achieved, is somewhat unpredictable from initial conditions (hence the term *a priori*).

The ability to fulfill mission responsibilities after the engagement is computed according to a smoothed step function (the border function) that breaks at the point where predicted losses reach one friendly aircraft. Other terms for predicted fuel state and time loss are present in a rudimentary form.

Design Element 20-3: Evaluate the Projected Candidate Alternative

Subroutine *aeval4* evaluates the projected candidate alternative. The principal component of the alternative score is the net value killed: *kill*aggfac-risk*. The factor *aggfac* is the flight leader's aggressiveness factor and influences his relative perception of the utility of destroying hostiles.

Scoring of flight postures uses the value function:

$$kill(aggfac) - risk + vmisn(fmisn)$$

where *aggfac* is an aggressiveness factor that may be set from input data for each flight, *fmisn* is a variable varying from zero to one which indicates the ability to continue the mission if the posture is adopted, and *vmisn* is the user input value of the mission.

The *aeval4* routine assigns a basic flight posture score based upon the ability to complete the mission, weighted by the value of the mission, the excess fuel resulting from the current

posture, weighted by the value of excess fuel, and the time taken by the current candidate posture, weighted by the value of time. This score is then adjusted by factors such as the anticipated kills, weighted by the flight's aggressiveness factor, the presence or absence of GCI vectoring inputs, and a range factor that rewards selecting a short-range posture if hostiles are close and a long range posture when hostiles are distant. Finally, a hysteresis factor is added that rewards selection of the same posture as was selected last time.

Design Element 20-4: Choose a Flight Posture Alternative

Subroutine *pkactn* is the executive decision-making routine which selects the highest scoring flight posture alternative. The algorithm loops through appropriate candidate alternatives and selects the highest scoring one.

Design Element 20-5: Implement Actions of Chosen Alternative

Subroutine *akshn4* implements the chosen alternative. The flight leader's mission, offensive, and defensive multipliers are set according to the selected alternative. These multipliers implement the flight posture alternative by influencing the value functions of other decisions lower in the decision hierarchy.

2.20.3 Functional Element Software Design

This section contains the software design necessary to implement the design requirements and the design approach defined in the preceding sections. The first subsection describes the subroutine hierarchy and describes how the subroutines work to make the flight posture decision. Subsequent subsections contain functional flow diagrams and describe all important operations represented by each block in the diagrams.

Flight Posture Decision Subroutine Hierarchy and Description

The major routines comprising the flight posture decision algorithm and their purpose are given below with the indentation of the routine name used to indicate the level of the routine within the calling tree.

modsel- pilot decision executive
- *pkactn* - generalized value-driven decision executive
  - *aslc4i* - initializes flight posture alternative generation procedure
  - *aeva4i* - initializes flight posture alternative candidate evaluation procedure
  - *aslct4* - generates candidate flight posture alternative
    - *alt41* - generates mission flight posture alternatives
  - *aproj4* - projects candidate flight posture alternative
    - *tloss* - evaluates total losses for a flight posture alternative
  - *aeval4* - evaluates candidate flight posture alternative
- *akshn4* - implements flight posture decision
- *gcitac* - changes the flight posture and tactic per the GCI message

Figure 2.20-2 presents this same information in standard calling-tree format.

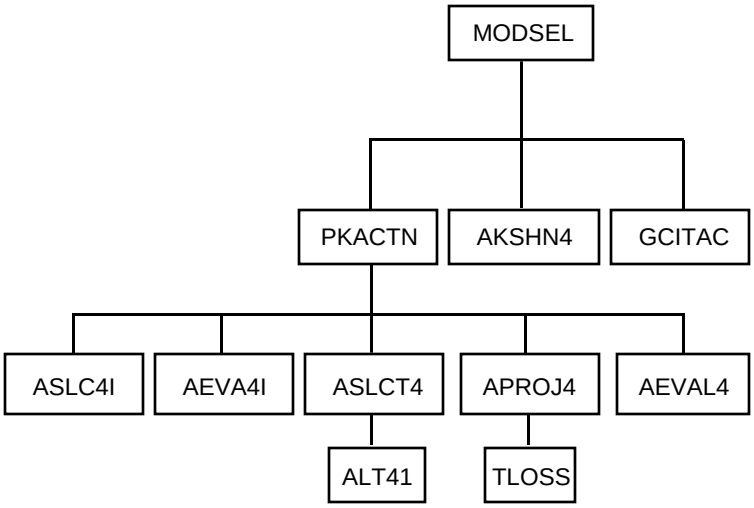

FIGURE 2.20-2. Flight Posture Decision Calling Tree.

A number of secondary subroutines used in the flight posture decision are defined below. These are not discussed in detail in this document, but are defined here as an aid to the reader.

capupd	Updates pilot's knowledge of his location on a CAP leg.
mppud	Updates pilot posture weapon parameters.
mppudi	Performs initialization for subroutine *prdexe*.
pcode	Production rules routine. May contain standard rules and/or user defined special handlers.
prdexe	Initializes and executes standard (level 0) production rules.
setlev	Ensures subordinate decisions are performed when needed.
setspt	Sets select pointer array for decision level.
svpred	Predicts state vectors a time *dt* into the future.
thrlim	Inhibits afterburner usage if an IR missile threat is perceived.
valsth	Determines the hostile with the highest average offensive score.
valsti	Initializes maneuver value components.

A number of utility subroutines are also used in the flight posture decision. These are not discussed in detail in this document, but are defined below as an aid to the reader.

border	Mathematical step function used in value functions.
cauchy	Mathematical bell-shaped function used in value functions.
ckrngi	Checks if a variable is within allowed range.
chralt	Formats an alternative descriptor word into a string.
daltpt	Deletes an alternative description.

delalt	Deletes an alternative description.
dot	Mathematical dot product function.
gcaprt	Retrieves CAP route data into a common block.
getarm	Returns information about the longest range weapon onboard specified aircraft.
grdrc	Retrieves radar characteristic data into a common block.
grdrs	Retrieves radar status data into a common block.
indalg	Returns the index of an alternative.
indg	Finds an alternative descriptor word's index.
indpk	Packs arguments into an alternative descriptor word.
indupk	Unpacks an alternative descriptor.
inlstv	Retrieves a record stored in list memory in V-format.
inlsva	Retrieves a portion of a list memory record.
inlsvd	Retrieves a record stored in list memory in V-format and automatically frees up list memory.
int_set	Transfers a bit pattern from a real to an integer variable.
lbit	Checks setting of specified bit in bit string.
match	Checks if a specified element matches a list element.
movalt	Copies an alternative descriptor.
nabort	Error handler - prints error message and aborts program.
olistv	Stores a record into list memory in V-format.
qguass	Fast and accurate generator of gaussian variates.
ramp	Mathematical ramp function.
rload	Performs typeless data transfers between integer and real.
sepa	Finds angle between two 3d vectors.
setfrm	Sets up the friendly and hostile formation data structures.
srch	Binary search of a real, ascending array.
stri	Converts a integer to a left-justified string.
tmstrt	Restarts a simulation timer (used for diagnostic profiling).
xmag	Finds magnitude of a 3d vector.
xmit	Copies data from one array to another.
xmitb	Copies one array to another in reverse order.
vsub	Performs 3-vector subtraction.

The principal data structures (common blocks) involved in the flight posture decision are described below.

/*altern*/	Holds parameters used in selecting alternatives.
/*althld*/	Holds the currently selected decision alternative.
/*altlst*/	Holds information about alternative descriptors.
/*decord*/	Specifies decision level hierarchy.
/*mind2*/	Holds value elements for each pilot.
/*mind3*/	Holds the pilot's assessment of relationships with other entities.
/*mind4*/	Holds mental model situational variables.
/*mind5*/	Holds values of assessed situational risks and expected kills.

Subroutine *modsel* is the executive that handles most of the flight leader and pilot decision-making functions. Specifically, it controls the flight leader posture and tactics decisions, the pilot posture decisions, the maneuver decision, and the weapon firing decision.

The logic of subroutine *modsel* consists primarily of deciding what decision level next requires a decision, and then calling *pkactn* and *akshn*N, (where N is a numeric decision level) to make and to implement the decision at that level. N = 4 for the flight posture decision. Some special conditions, such as whether the conscious entity makes a decision of a particular type (is he a flight leader?) are explicitly considered. The *declev* subroutine handles decision requirements caused externally or by the time interval since the last decision at this level by this pilot. In addition, *modsel* performs initialization functions by calling *valsti* and *svpred*. Before making decisions, subroutine *prdexe* is called to allow production rule biasing of the pilot's decisions when specified by the user.

Subroutine *pkactn* is the true decision executive at a given decision level. It first enters a loop that calls subroutines *aslct, aproj*, and *aeval* (these are generic subroutine names; actual entry names are passed as arguments to *pkactn*). These subroutines generate, project, and evaluate the alternative. The first pass through the loop will only consider those alternatives biased in production rules. The second pass considers all allowed alternatives if the first pass generates nothing. Execution of the second pass can be inhibited by the user by setting an operational mode array element, *ouemod(19)*, in the **SCNRIO** input data file.

A major secondary function of *pkactn* is to integrate the biases of the production rule system with the value system. There is also a fair amount of additional code but it is almost entirely related to maintaining a list of the best-scoring alternatives. There are two reasons for this. First, when debugging; it is often useful to see which alternatives were almost picked. A secondary function is to permit the addition of small random changes to the value scores of the alternatives. This permits the simulation of a degraded pilot decision-making capability, as when a pilot is under extreme stress. Pointers to the best alternatives are kept in the array *locval*, and their value scores in array *altval*. When only one alternative is generated for consideration, *pkactn* bypasses the scoring mechanism and assigns a score in *altval* for that alternative.

When *pkactn* returns, it will have placed the description of the selected alternative in the /*althld*/ common block, for use by the *akshn* routine.

This section describes the organization of the generic alternative generation, projection, and evaluation routines called by *pkactn*. The actual names of the subroutines are formed by appending a suffix to the generic names indicating the decision level (= 4 for the flight posture decision, which produces *aslct4*, *aproj4*, *aeval4*, *akshn4*). In addition, the subroutines that generate individual alternatives, which all begin with '*alt'*, have a second suffix that indicates the index of that alternative within the alternative set, i.e., *alt41*, *alt42*, etc. Occasionally, a third suffix is used to further subdivide alternatives.

The purpose of the *aslct* routine is to generate a different alternative each time it is called, or return a flag indicating that the available set of alternatives has been exhausted. Because the set of alternatives considered occasionally changes as the Brawler model is developed, it is not sufficient to label each alternative by a simple numerical index; such a system would greatly reduce the flexibility available for adding new alternatives. Instead, alternatives are labeled with a hierarchical notation consisting of four indexes, referred to as the index set (*ilevel*, *kalt*, *icall*, *lcall*). The outermost variable *ilevel* denotes the decision level for which the alternative is a course of action (4 for the flight posture decision). *Kalt* denotes the most general kind of alternative. The variable *icall* is used to further differentiate alternatives when several have the same *ilevel* and *kalt* values. The variable *lcall* is used in those cases where a breakdown beyond the *icall* level is required. It is currently used only in the specification of flight tactics.

*aslct*N (N corresponds to *ilevel*) uses the *kalt* index from the index set to break up the alternative enumeration into more easily manageable parts. For each *kalt* value, each call to *aslct*N causes it to call an *alt*NK subroutine (N=*ilevel*, K=*kalt*) until the latter returns *more* = .false., indicating exhaustion of alternatives. The *aslct* routine is responsible for setting *icall* to zero prior to the first time each *alt* routine is called (for each consciousness event). This serves to trigger internal initialization by the *alt* routine. The *alt* routine returns the *icall* value of the generated alternative.

The *aproj*N subroutine is used to project or predict the consequences of adopting an alternative without putting a value on the consequence. For those decisions that are not actually value-driven, the *aproj*N subroutine is a dummy.

The *aeval*N subroutine places a value on the predicted consequence of a candidate alternative. The *akshn*N subroutine is used to actually implement the action that *pkactn* has selected. This implementation may include altering the value parameters of a conscious pilot in order to influence lower level decisions.

In addition to the above decision routines, there are also initialization entry points to the *aslct*N and *aeval*N subroutines. Actual subroutine *aslct*N has initialization entry *aslc*N*i* and *aeval*N has entry *aeva*N*i*. The *aslc*N*i* entry is responsible for setting *kalt*=1 and *icall*=0 so that the first call to *aslct*N will function properly. Other action is optional and specialized to the nature of the particular decision level. The *aeva*N*i* entry either performs specialized processing, or is a dummy entry point.

Subroutine MODSEL

Subroutine *modsel* is the executive subroutine of the entire pilot decision process of which the FPDFE is included. Figure 2.20-3 is the functional flow diagram which describes the

logic used to implement the portion of *modsel* relevant to implementing the flight posture decision. The blocks are numbered for ease of reference in the following discussion.

Block 1: Value statistics used for the maneuver decision are initialized with a call to subroutine *valsti*.

Block 2: Test if missile mode = 2, ready to fire.

Block 3: If a missile is ready to fire set the projection interval *tproj3*, used to project other aircraft in the simulation, to 2 seconds.

Block 4: If a missile is not ready to fire find the minimum range *rngmin* to the closer of either a hostile aircraft or hostile missile and then..... The availability of guns is determined from the weapons stores list.

Block 5: .....compute *tproj3* as a ramp function of *rngmin*.

Block 6: Perform a constant acceleration projection of all other aircraft *tproj3* seconds into the future using subroutine *svpred*.

Block 7: Retrieve the current aircraft's radar characteristic and status variables using accessor subroutines *grdrc* and *grdrs* respectively.

Block 8: Initialize all alternative consideration flags to false.

Block 9: Set mental model index *ppmiac* with subroutine entry *mppudi* prior to ensure that it references the correct target aircraft.

Block 10: Execute the routine (level 0) production rules with a call to subroutine *prdexe*.

Block 11: Initialize the production rule fire control range variable *pr_ctrl_rng* to 10 nautical miles.

Block 12: Execute the fire control threshold production rule handler with a call to subroutine *pcode*(15).

Block 13: Test if the current mission is a CAP station mission. If it is not then skip the CAP mission update logic (skip to block 24).

Block 14: If a CAP station mission unpack the tactics decision alternative descriptor *altd1* with a call to subroutine *indupk*.

Block 15: Test if the tactics alternative is a GCI tactic.

Block 16: If the tactic alternative is not a GCI tactic perform a CAP mission update by retrieving the CAP route via a call to subroutine *gcarpt* and then.....

Block 17: .....updating the pilot's knowledge of his location on the route with a call to subroutine *capupd*.

Block 18: If the tactic alternative is a GCI tactic test if it is a GCI vector tactic. If it is then skip the CAP mission update logic (skip to block 24).

Block 19: If the GCI tactic is not a vector tactic test if it is a GCI drag tactic. If it is not then.....

Block 20: .....abort the program with a call to subroutine *nabort* since it is an unknown GCI tactic.

Block 21: If the GCI tactic is a drag tactic test if the desired speed set by the GCI is non-zero. If non-zero then skip the CAP mission update logic (skip to block 24).

Block 22: If the GCI tactic is not a drag tactic perform a CAP mission update by retrieving the CAP route via a call to *gcaprt* and then.....

Block 23: ..... updating the pilot's knowledge of his location on the route with a call to subroutine *capupd*.

Block 24: Update the pilot posture weapon parameters with a call to subroutine *mppud*.

Block 25: Test if projection interval *tproj3* is not equal to 2 and if the missile mode is equal to 2 (ready to fire).

Block 26: If the conditions of block 25 are met set *tproj3* to 2 and then.....

Block 27: .....project all other aircraft in the simulation ahead by *tproj3* seconds with a call to subroutine *svpred*.

Block 28: Set the alternative format indicator *iactn* to 0 indicating no format.

Block 29: A call to subroutine *thrlim* limits the throttle if an IR missile threat is perceived.

Block 30: Set loop counter *jlevel* to 0.

Block 31: Test if *jlevel = nlevel*, the number of decision levels. This is the starting point of the decision loop which loops over the various decisions to be made in this routine.

Block 32: If *jlevel = nlevel*, invoke the target conflict production rules handler with a call to subroutine *pcode*(7) and then return control to the calling routine *conevt*.

Block 33: If *jlevel* is not equal (is less than) *nlevel* increment *jlevel*.

Block 34: Set the decision level *ilevel* to the value of the decision level order array at index *jlevel*, *decord*(*jlevel*).

Block 35: Set the alternative select pointer for decision level *ilevel* with a call to subroutine *setspt*.

Block 36: Set the alternative format indicator to 0 indicating no format.

Block 37: Test if all timers are active.

Block 38: If all timers are active reactivate the clock with a call to subroutine *tmstrt*.

Block 39: A computed goto statement acts as a switch dependent upon *ilevel* to select the decision to be processed. If *ilevel* is 4 then the algorithm branches to the flight posture decision process. If it is another value then some other decision process is performed. The routine will visit each decision process in the switch statement before the routine exits.

Block 40: Test if the decision maker is a surface to air missile (SAM) site. If yes, then control branches to block 46.

Block 41: If the entity is not a SAM site (is an aircraft) test if the current aircraft is the flight leader and that all the prerequisites are satisfied for making the flight posture decision. If not then control branches to block 45.

Block 42: If the block 41 conditions are met call subroutine *pkactn* to make the flight posture alternative selection.

Block 43: Test if *iactn* is not equal to 0. A nonzero value indicates that some alternative format is indicated by *iactn*.

Block 44: If *iactn* is not 0 implement the selected alternative with a call to subroutine *akshn4*.

Block 45: Change the flight posture decision for any GCI messages with a call to subroutine *gcitac*.

Block 46: Test if all timers are active. If not then branch back to the start of the decision process at block 31.

Block 47: If all timers are active stop them with a call to subroutine *tmstop*. Then branch back to the start of the decision process at block 31.

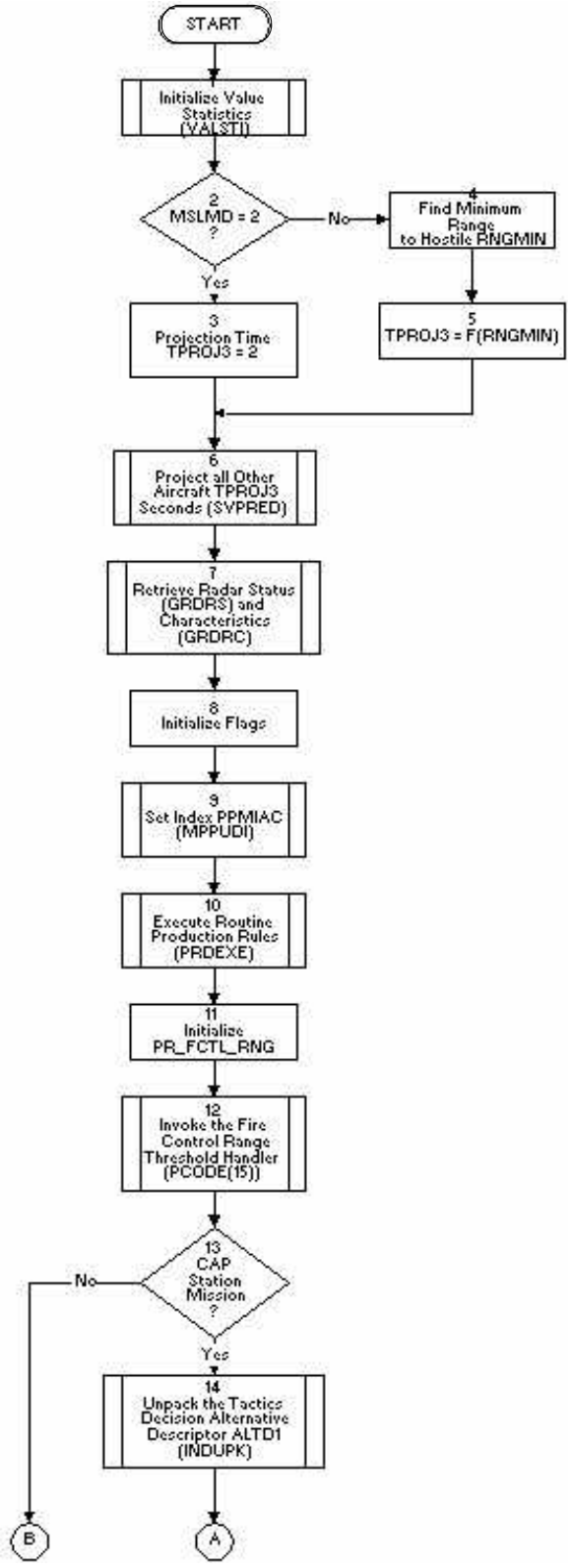

FIGURE 2.20-3. MODSEL Functional Flow Diagram (Page 1 of 4).

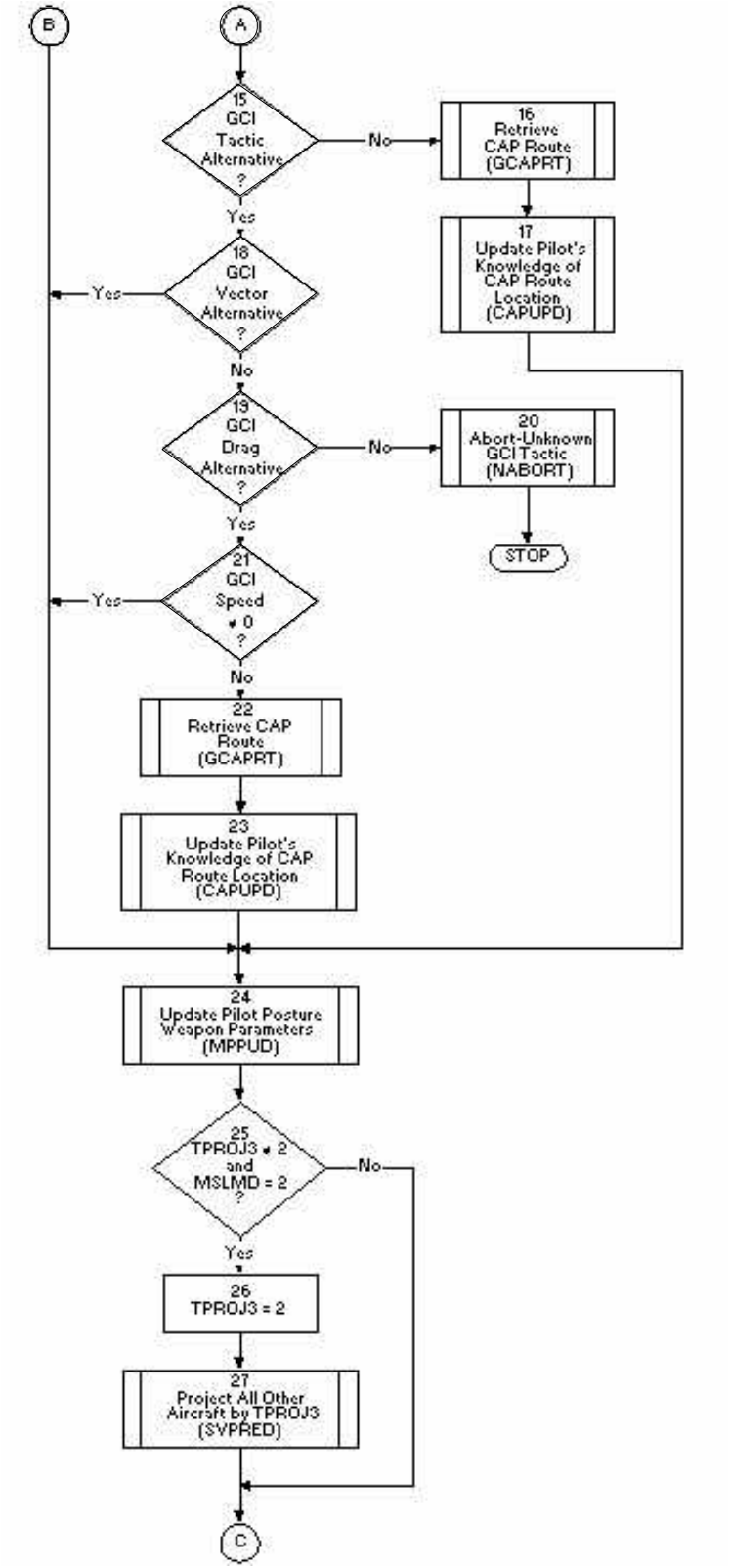

FIGURE 2.20-3. MODSEL Functional Flow Diagram (Page 2 of 4).

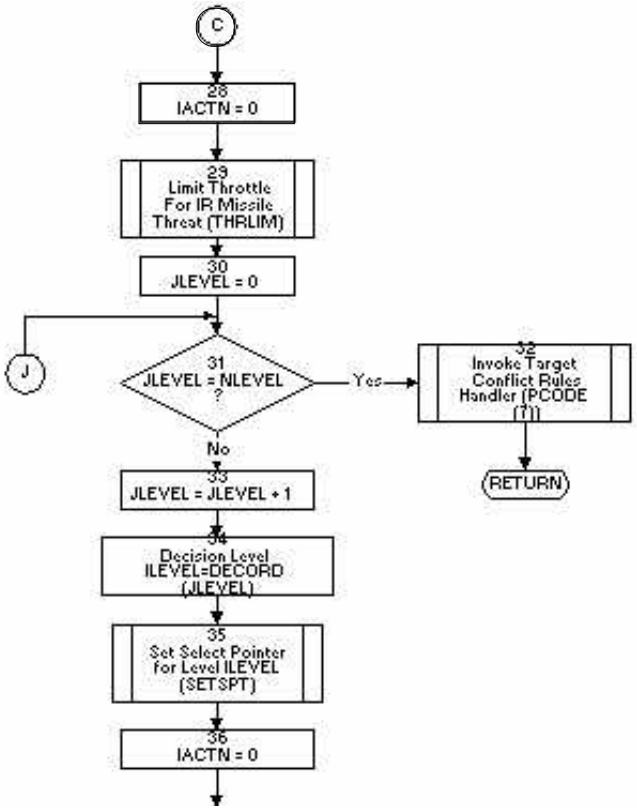

FIGURE 2.20-3. MODSEL Functional Flow Diagram (Page 3 of 4).

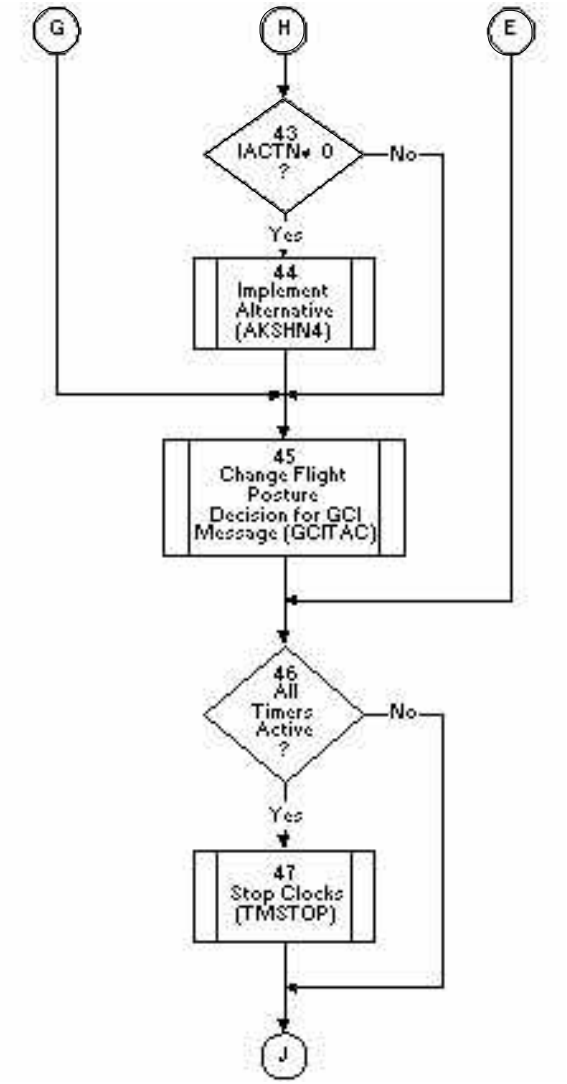

FIGURE 2.20-3. MODSEL Functional Flow Diagram (Page 4 of 4).

Subroutine PKACTN

Subroutine *pkactn* is the executive procedure for value-driven decision selections. Figure 2.20-4 is the functional flow diagram which describes the logic used to implement *pkactn*. The blocks are numbered for ease of reference in the following discussion.

Note that five subroutine names are passed in as arguments to *pkactn*, which is a FORTRAN feature not commonly encountered. This means that flight posture specific routines referred to in the following discussion, such as *aslct4*, appear in the actual code in *pkactn* as generic names, such as *aslct*.

Block 1: Setting *altpas* to 1 tells the alternative generation routine *aslct4* to not consider unbiased (by production rules) alternatives.

Block 2: Subroutine *setlev* ensures that all subordinate decisions will be reexamined after making the flight posture decision.

Block 3: The flight posture alternative generation procedure is initialized by a call to entry *aslc4i* of subroutine *aslct4*.

Block 4: The flight posture alternative candidate evaluation procedure is initialized by a call to entry *aeva4i* of subroutine *aeval4*.

Block 5: The bias strength is computed as a function of the production rule bias strength variable *biastr*.

Block 6: The flag *gnrate* is set true which tells the algorithm to generate an alternative.

Block 7: Test flag *gnrate* to determine if an alternative is to be generated.

Block 8: If *gnrate* is true, an alternative is generated by a call to subroutine *aslct4*.

Block 9: If *gnrate* is false, the second alternative generated is restored into array *althld*.

Block 10: Test on whether there are more alternatives to consider.

Block 11: If there are no more alternatives to consider, test if only one alternative has been generated.

Block 12: If only one alternative was generated, store it in the *althld* array.

Block 13: Initialize the alternative value variables *altvlx* and *altvly* to 20.

Block 14: If there are more alternatives to consider (from block 10 test), test if only one alternative was generated.

Block 15: If only one alternative was generated and there are more to generate, save the first alternative in a temporary location.

Block 16: If there was not one alternative generated, test if two alternatives were generated.

Block 17: If two alternatives were generated, then save the second one in a temporary location and.....

Block 18: ..... restore the first alternative generated in array *althld*.

Block 19: Set *gnrate* to false.

Block 20: If the number of alternatives generated is not one or two then set *gnrate* to true.

Block 21: Call subroutine *aproj4* to project the candidate flight posture alternative.

Block 22: Test if the alternative is a duplicate of a previously considered alternative and if it is not positively biased by the production rules. If these conditions are both true, then

this alternative is effectively canceled (by not continuing the processing of it) and the algorithm returns to block 7 to decide whether to generate another alternative.

Block 23: If the test of block 22 fails, then the alternative candidate is still valid and it is evaluated by subroutine *aeval4* which assigns the alternative a value *altvly*.

Block 24: Test if the production rule bias strength = 0.

Block 25: If the production rule bias strength is not 0, then a gaussian variate is generated by function *qgauss* which is then used to.....

Block 26: ..... compute the alternative's value *altvlx* along with the bias strength.

Block 27: If the production rule bias strength is 0 then a gaussian variate is generated by function *qgauss* which is then used to.....

Block 28: ..... compute the alternative's value *altvlx* without the bias strength.

Block 29: Test if the alternative value *altvlx* is above the minimum cutoff value *vcut*.

Block 30: If the alternative value is below the cutoff then delete the alternative by a call to subroutine *delalt*.

Block 31: If the alternative value is at or above the cutoff then test if the number of alternatives evaluated, *nalt*, is not equal to 0.

Block 32: If *nalt* is 0, then set flag *smalst* to false and flag *largst* to true which indicates that the current alternative candidate is the highest valued alternative (since it is the only alternative evaluated).

Block 33: If *nalt* is not 0, then there is more than one alternative generated, and flags *smalst* and *largst* are logical functions of *altvlx* and *altval*.

Block 34: Test if *nalt* equals the maximum allowed number of evaluated alternatives.

Block 35: If *nalt* is at maximum, test the *smalst* flag.

Block 36: If *smalst* is true, then the current alternative is the lowest valued one and is to be deleted by calling subroutine *delalt.*

Block 37: If *smalst* is false, *nalt* is decremented.....

Block 38: .....and the first alternative on the list is deleted by a call to *daltpt*.

Block 39: If *nalts* is not equal to the maximum allowed (from block 34), store the current alternative in list temporary list memory with a call to subroutine *olistv*.

Block 40: Test the *smalst* flag.

Block 41: If *largst* is false and *smalst* was also false, then the current alternative is between the lowest and highest scoring so far and is inserted in order into the list of candidates.

Block 42: *nalt* is incremented for the alternative added in block 41.

Block 43: If *largst* is true then the current alternative is the highest scoring to this point. *nalt* is incremented and.....

Block 44: .....the alternative is inserted into the ordered list of candidate alternatives.

Block 45: Test if the current alternative score *altvlx* (which is the highest scoring alternative) is not negative.

Block 46: If *altvlx* is negative, calculate the threshold score *vcut* as *altvlx*/*vcutf*.

Block 47: If *altvlx* is positive, calculate the threshold score *vcut* as *altvlx***vcutf*.

Block 48: This is a loop statement for looping over the alternative score list *altval*(*i*) in ascending order of scores (since it is an ascending-ordered list). Blocks 49 and 50 are included in the loop.

Block 49: Component of the loop sequence of block 48. Tests if the currently indexed score *altval*(*i*) is less than the threshold *vcut*.

Block 50: Component of the loop sequence of block 48. Given that the currently indexed alternative *altval*(*i*) is below the threshold *vcut*, delete the alternative by a call to subroutine *daltpt*.

Block 51: Given that the currently indexed alternative was above the threshold *vcut*, test if the final value of index *i* of the previous loop is greater than 1 which tests to see if one or more alternatives were deleted from the list.

Block 52: If one or more alternative were deleted, compress the list.

Block 53: If no alternatives were deleted from the list, test to see if there are more alternatives to be generated. If yes then go back to block 7 to start the process of generating another alternative.

Block 54: If are no more alternatives to be generated test if the number of alternatives evaluated, *nalt*, is above zero.

Block 55: If *nalt* is above zero, test if only positively biased alternatives are to be considered and if routine *aslct4* skipped unbiased alternatives during this pass through the list of alternatives (*altpas* = 1).

Block 56: If the test above (block 55) is true then *altpas* is set to 2, meaning unbiased alternatives will not be skipped, and return to block 2 to process unbiased alternatives. The effect of this is that if the first pass through the alternatives only considered biased alternatives and none were generated, then a second pass must be made that looks at all alternatives.

Block 57: If the conditions of block 55 are not met then *iactn*, the alternative format indicator in common block /*althld*/, is set to 0 (indicating no format) and the routine returns to the calling routine *modsel*.

Block 58: If the number of alternatives evaluated, *nalt*, is below 0, test if the decision level (*ilevel*) is at level 3.

Block 59: If the current decision level is 3, compute the average of each maneuver value component with a call to subroutine *valst*.

Block 60: This is a loop statement for looping over all alternatives which have been evaluated (*nalt* alternatives). Blocks 61 through 63 are included in the loop.

Block 61: This is a component of the block 60 loop. Test if the currently indexed alternative is the last on the list and therefore highest valued one.

Block 62: This is a component of the block 60 loop. If the current indexed alternative is not the highest valued one, delete it from the list with a call to subroutine *daltpt*.

Block 63: Test if there are more alternatives for the loop to operate on. If yes, return to the top of the loop.

Block 64: If there are no more alternatives for the loop to operate on, copy the remaining alternative (highest valued) on the candidate list from temporary storage to the *althld* array in common block /*althld*/ with a call to subroutine *inlsvd*.

Block 65: Also store the highest valued alternative in array *cactn* of common block /*mind2*/ then return to the calling routine *modsel*.

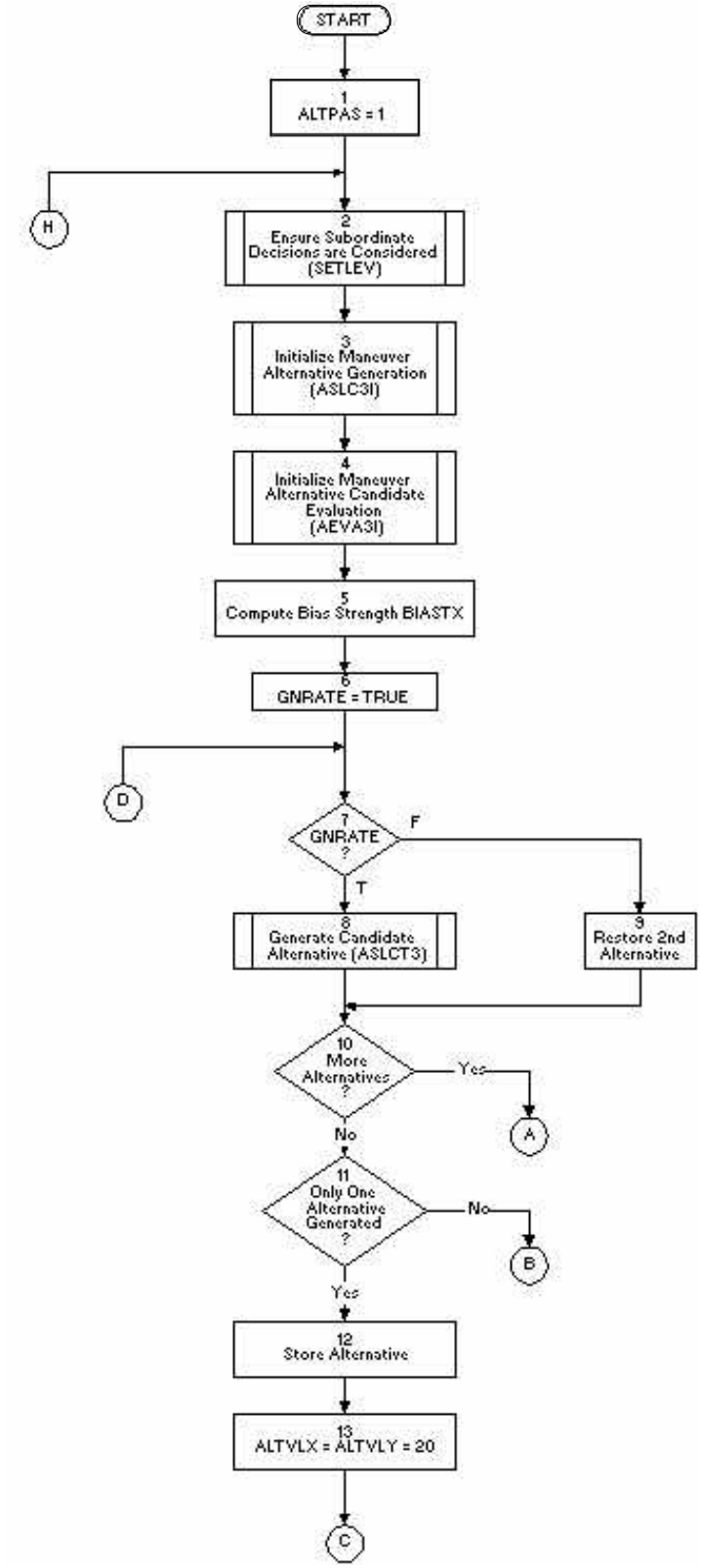

FIGURE 2.20-4. PKACTN Functional Flow Diagram (Page 1 of 5).

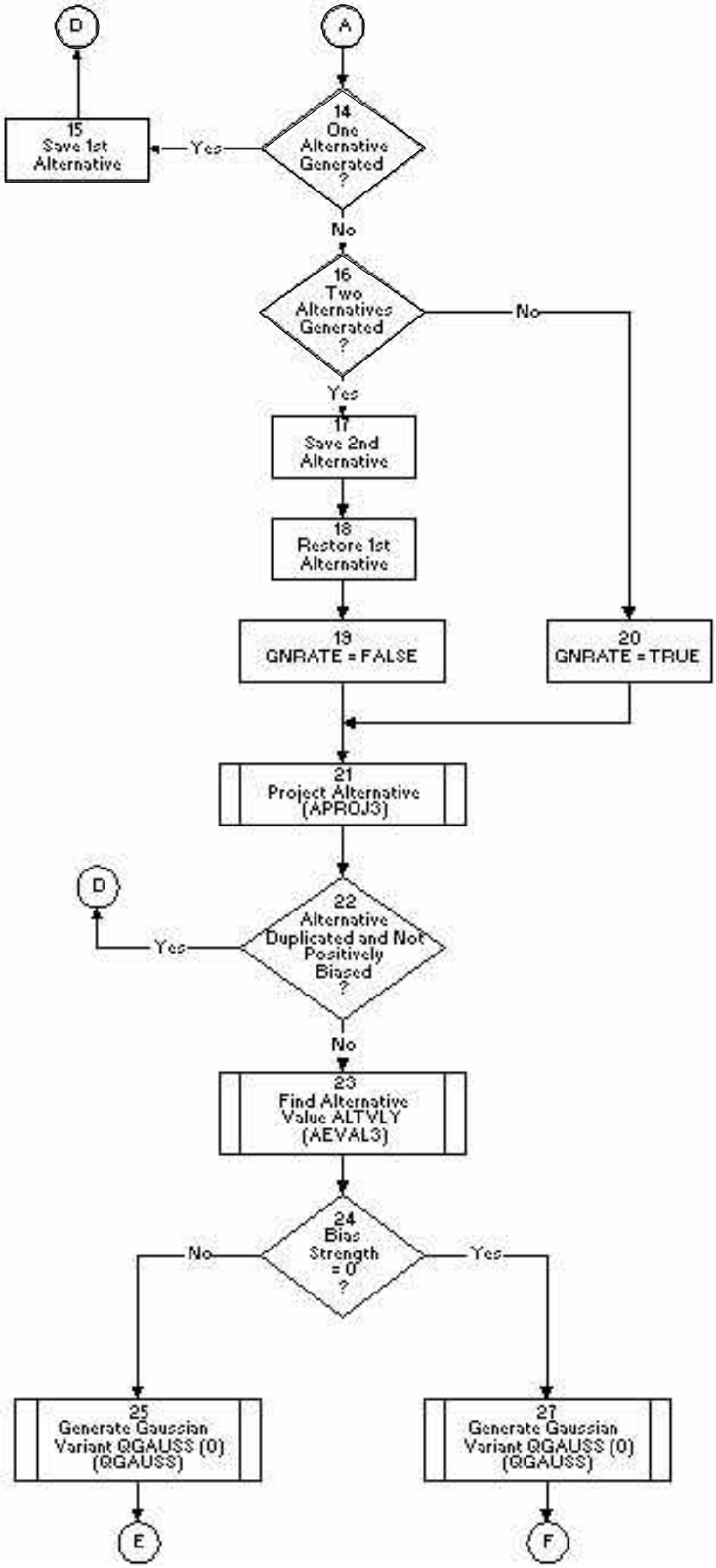

FIGURE 2.20-4. PKACTN Functional Flow Diagram (Page 2 of 5).

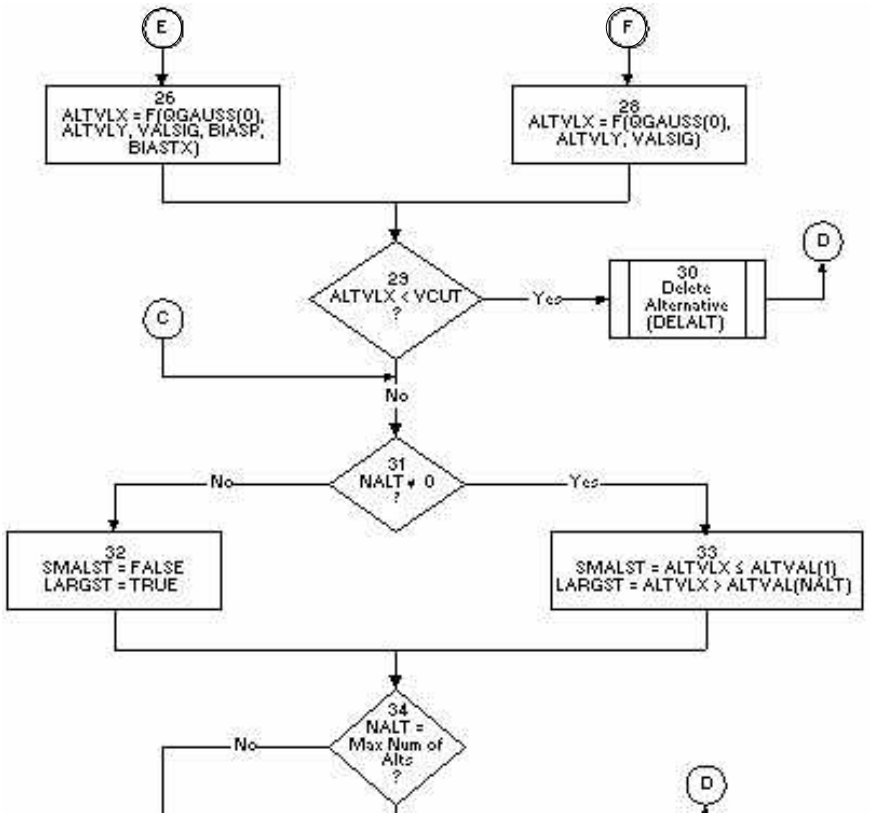

FIGURE 2.20-4. PKACTN Functional Flow Diagram (Page 3 of 5).

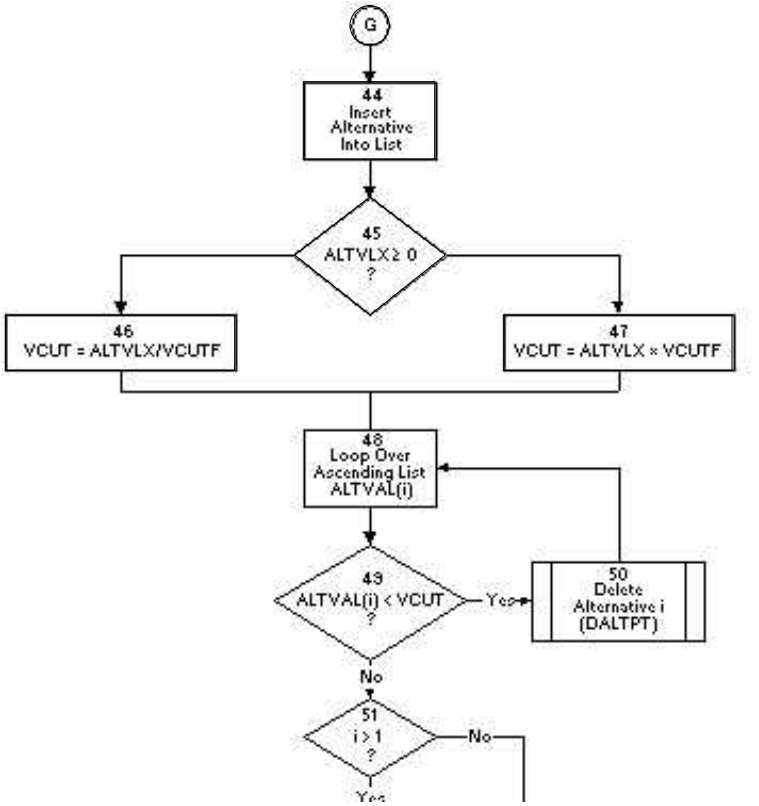

FIGURE 2.20-4. PKACTN Functional Flow Diagram (Page 4 of 5).

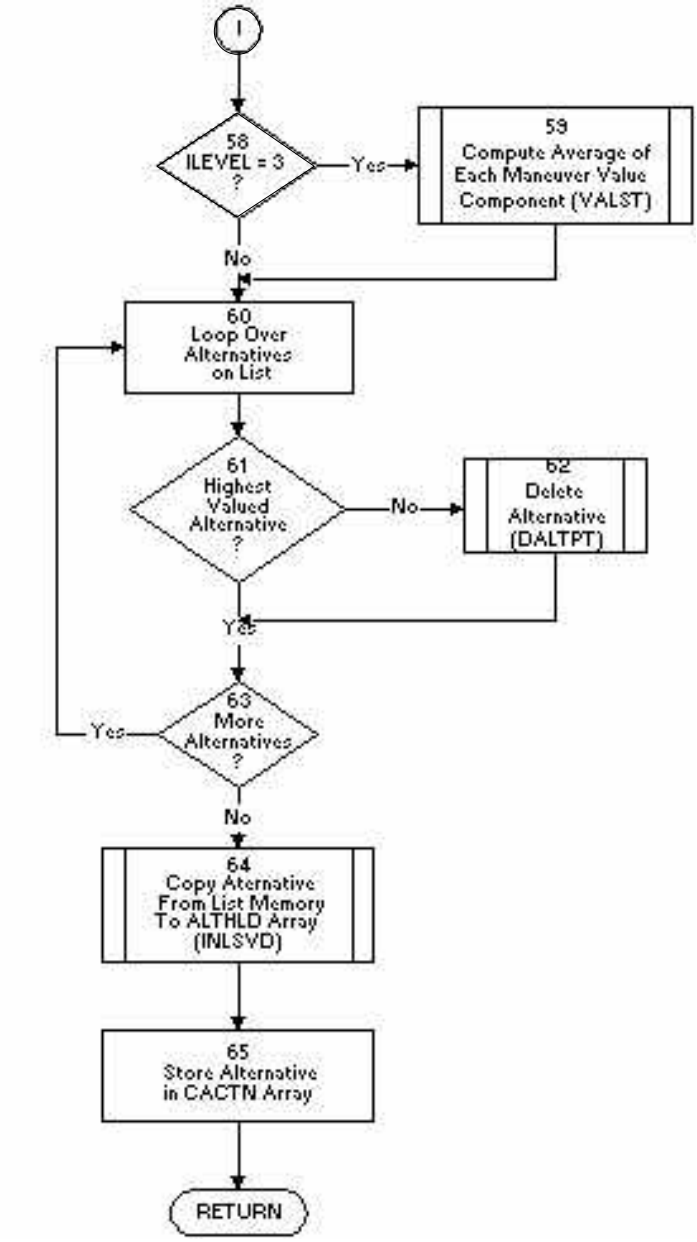

FIGURE 2.20-4. PKACTN Functional Flow Diagram (Page 5 of 5).

Subroutine ASLC4I

Subroutine (entry) *aslc4i* initializes the alternative selection process. Figure 2.20-5 is the functional flow diagram which describes the logic used to implement *ascl4i*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Set the alternative kind indicator *kalt* to 1, the first alternative kind.

Block 2: Set the alternative kind subtype indicator *icall* to 0, the first subtype.

Block 3: Set the "no bad guy aircraft" indicator to false.

Block 4: Beginning of a loop statement over all hostile aircraft (*nbg*, which stands for "number of bad guys"). The loop encompasses blocks 4 through 6.

Block 5: Test if the currently indexed hostile entity is an aircraft. If yes, then branch out of the loop to block 8.

Block 6: If more hostile entities, go back to the top of the loop, otherwise exit the loop.

Block 7: If the loop terminates due to no more hostiles, then this statement is executed next which sets the "no bad guy aircraft" indicator to true.

Block 8: Test if there are any previous communication orders by seeing if old order pointer *oldop* is non-zero.

Block 9: If no previous communication orders exist, set the old order pointer *oordp* to 0, and the old order time variables *oordt* and *oldotm* to a large negative value (-1000).

Block 10: If a previous communication order does exist set the old order variables in blocks 10 though 12. Set *oordp* to the value of the order pointer *oldop* and set *oordt* to the value of previous order time *oldotm*.

Block 11: Copy alternative descriptor *cactn*(1,1) to *ocact1*(1) with a call to subroutine *movalt*.

Block 12: Copy alternative descriptor *cactn*(1,4) to *ocact4*(1) with a call to subroutine *movalt*.

Block 13: Test if any previous tactics orders exist by seeing if old tactics pointer *otacp* is non-zero.

Block 14: If an old tactics order exists store it into array *opplan* using subroutine *inlstv*.

Block 15: Set the old tactics order pointer to -1.

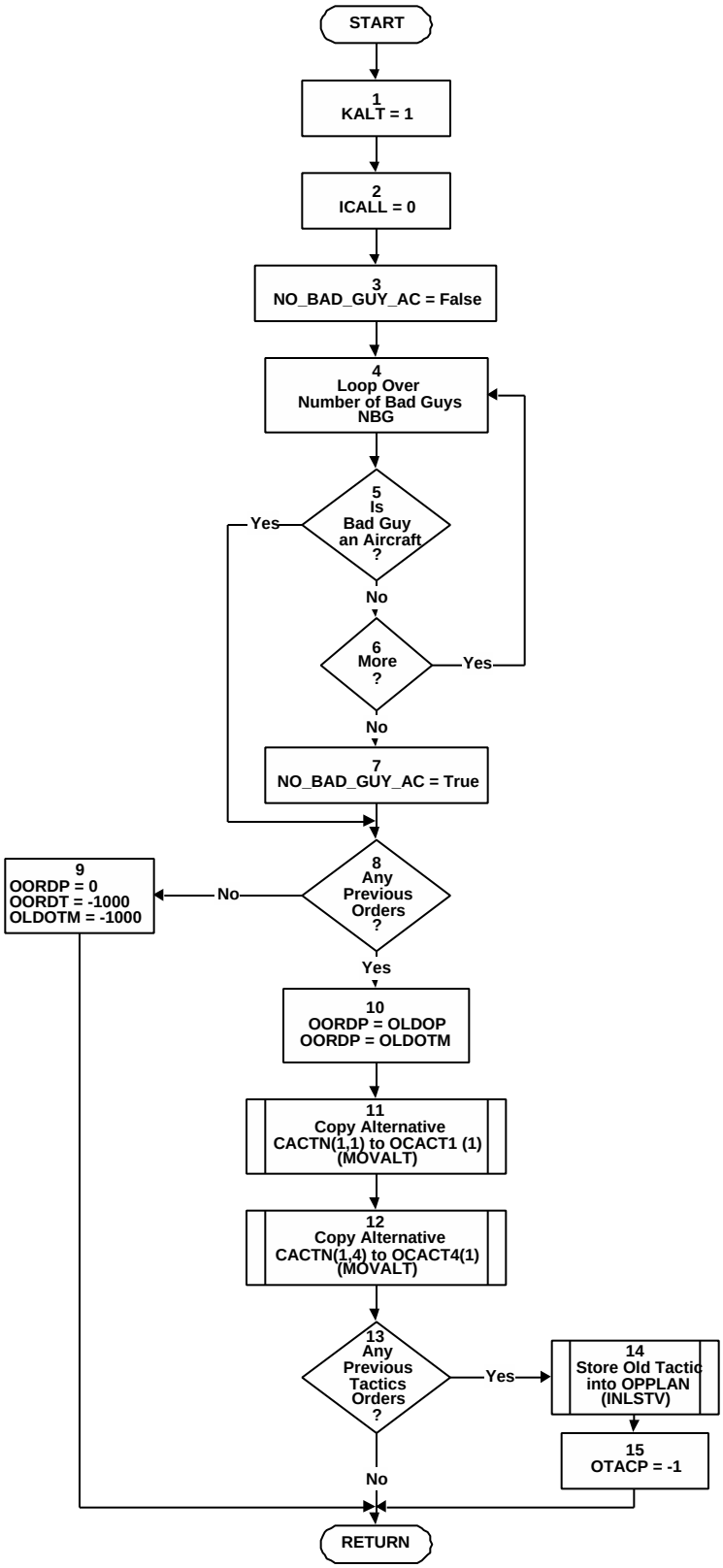

FIGURE 2.20-5. ASLC4I Functional Flow Diagram.

Subroutine AEVA4I

Subroutine entry *aeva4i* performs initialization for the alternative evaluation procedure which represents the level of satisfaction of the weapon envelope. Figure 2.20-6 is the functional flow diagram which describes the logic used to implement *aeva4i*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Determine the longest range weapon available to me with a call to subroutine *getarm*.

Block 2: Check that the weapon found in block 1 is a legitimate type with a call to subroutine *ckrngi*.

Block 3: TestDecision on if the longest range weapon *kndbst* = 0, which indicates a gun.

Block 4: If *kndbst* was 0 then set kndbst to 4 which also indicates a gun.

Block 5: Calculate maximum weapon range *rnguse* as the larger of the nominal weapon range *rngmax* and 4 nautical miles.

Block 6: Initialize *psavg*, the average probability of being seen, *vftot*, the total friendly value, and *risks*, the expected flight value lost if an engagement ensues, all to zero.

Block 7: Loop statement for looping over aircraft in my flight to assess risk factors and compute *psavg*. Blocks 7 through 12 are included in the loop.

Block 8: Find the aircraft index *iac* from list *lmyflt* for the current loop index.

Block 9: Increase *vftot* by the intrinsic value of aircraft *iac*

Block 10: Increase *psavg* as a function of *pseen*(*iac*), the estimated probability that a friendly aircraft has been detected.

Block 11: Increase *risks* as a function of the intrinsic value of aircraft *iac* and the probability that aircraft *iac* will survive.

Block 12: Test if more aircraft in my flight. If yes, branch back to the top of the loop (block 7).

Block 13: Divide *psavg* by the number of aircraft in my flight to get the final value of *psavg*.

Block 14: Test if any bad guy aircraft exist. If no, branch down to block 35. If yes, do blocks 15 through 34.

Block 15: Set up friendly and hostile formation data structures with a call to subroutine *setfrm*.

Block 16: Compute the position vector *dx* from the center of the friendly formation to the center of the hostile formation with a call to utility subroutine *vsub*.

Block 17: Compute the range to the hostile formation (*range*) from *dx* using function *xmag*.

Block 18: Compute the velocity vector difference *dv* between the friendly and hostile formation group velocity vectors using subroutine *vsub*.

Block 19: Compute the relative speed *spdtgt* between the hostile and friendly formations from *dv* using function *xmag*.

Block 20: Compute the range rate, *rdot*, by taking the dot product of *dx* and *dv* divided by *range*.

Block 21: Compute the angle *theta* between *dx* and the hostile formation group velocity vector minus 90 degrees using function *sepa*. This measures how close the hostile formation velocity is to perpendicular to the line of sight to the hostiles.

Block 22: Compute the effective weapon range *rngeff* as a function of *range*, *theta*, *rdot*, and *spdtgt*.

Block 23: Compute range factor *rngfac* as a function of *rnguse* and *rngeff*.

Block 24: Compute *riska*, the a priori expected risk, as a function of *vftot* and a function of *fratio*, the force ratio.

Block 25: Initialize to zero *kills*, the expected hostile value killed if an engagement ensues and *vhtot*, the total hostile value.

Block 26: Loop statement for looping over bad guy aircraft to assess kill factors. Blocks 26 through 30 are included in the loop.

Block 27: Find bad guy index *iac* from list *listh* indexed by the current loop index.

Block 28: Increase *vhtot* by the value of engaging aircraft *iac*.

Block 29: Increase *kills* by the probability aircraft *iac* will survive and the value of engaging aircraft *iac*.

Block 30: Test of more bad guys. If yes, branch back to the top of the loop. If no, continue execution at the next block.

Block 31: Test if *rngeff* is greater than or equal to *rngwpn*. If no, branch to block 33. If yes, continue execution at the next block.

Block 32: If *rngwpn* is smaller than *rngeff* calculate *psavg* as a function of *kills*, *rngeff*, *rngwpn*, and *detr0*, the nominal detection range by hostiles.

Block 33: Compute *kills* as a function of the current value of *kills*, *psavg*, and *vhtot*.

Block 34: Compute *killa*, the a priori expected kill value as a function of *vhtot* and a *cauchy* function of the inverse of the force ratio.

Block 35: If no bad guys exist, initialize *rngeff* to *xlarge* (the largest real number attainable by the system), *rngfac* to 1, and *risks*, *kills*, *riska*, *killa*, and *vhtot* all to 0.

Block 36: Compute *rngbon* as one-tenth the sum of *riska* and *killa*.

Block 37: Unpack alternative descriptor *altd1* using subroutine *indupk* then return control to the calling routine *pkactn*.

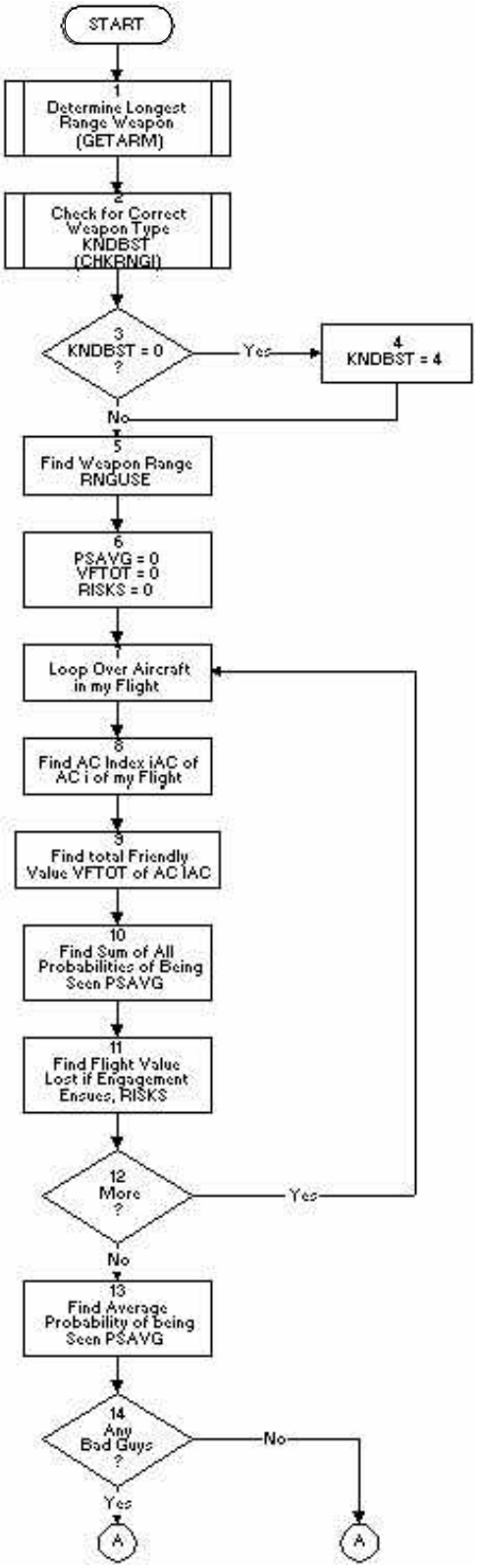

FIGURE 2.20-6. AEVA4I Functional Flow Diagram (Page 1 of 3).

FIGURE 2.20-6. AEVA4I Functional Flow Diagram (Page 2 of 3).

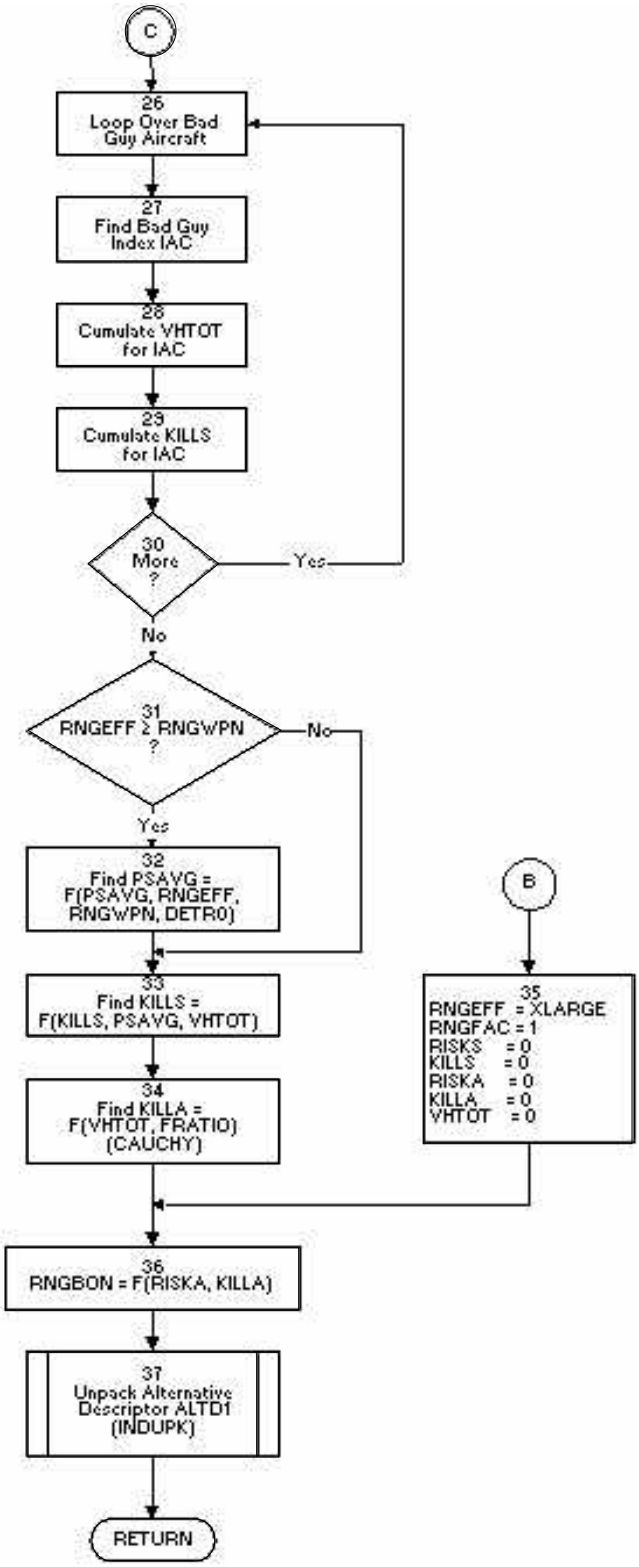

FIGURE 2.20-6. AEVA4I Functional Flow Diagram (Page 3 of 3).

Subroutine ASLCT4

Subroutine *aslct4* generates a candidate flight posture alternative. Figure 2.20-7 is the functional flow diagram which describes the logic used to implement *aslct4*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Check *kalt* range (between 1 and 8 inclusive) with subroutine *chkrngi* to ensure a valid alternative kind is to be processed.

Block 2: If *kalt* is 1 (is a mission posture alternative kind), then continue processing at block 3 to generate a mission flight posture alternative. Otherwise, continue at block 10 to process any of the other alternative kinds which are specified by *kalt* and *icall*.

Block 3: Generate the mission flight posture alternative with subroutine *alt41*.

Block 4: Test if an alternative was generated. If yes, continue at block 5. If no, continue at block 6.

Block 5: Set indicator 'more' to true to indicate to the calling routine (*pkactn*) that more alternatives are to be generated and then return to the calling routine (*pkactn*).

Block 6: Increment *kalt* by 1 to process the next alternative kind.

Block 7: Test if *kalt* = 7 (SAM site alternative kind). If not, continue at block 9. If yes, then continue at block 8.

Block 8: If a SAM site alternative kind, set 'more' to false indicating no more alternatives to process and the return to the calling routine *pkactn*.

Block 9: If the kind is not a SAM site, then set *icall*, the kind subtype, to 0 and branch back to block 2 to continue processing the next alternative.

Block 10: Begin processing of other (any but mission flight posture) alternative kinds by incrementing *icall* by 1 to the next subtype.

Block 11: Test if *icall* equals 2. If yes, branch to block 6. If no, continue at block 12.

Block 12: If *icall* is not 2, test if there are no bad guys and that *kalt* is not equal to 6, return to base alternative kind. If true, then branch to block 6. If false, continue at block 13.

Block 13: Get the index *ind* of the alternative using subroutine *indalg*.

Block 14: Test if alternative *ind* is to be considered by testing the value stored in *sptr*(*ind*). If not, return to block 11.

Block 15: Set *iactn*, format indicator, to 8 indicating flight posture format.

Block 16: Set the alternative length *lenalt* to the value stored in *lactn*(*iactn*).

Block 17: Pack the alternative defined by *kalt* and *icall* into descriptor word *altdsc* using subroutine *indpk*.

Block 18: Set alternative description array indices 4, 5, and 6 with 0, *fltp* (current flight posture indicator), and *kalt* respectively using subroutine *rload*.

Block 19: Test if *kalt* is equal to 4 (disengage) or 6 (return to base). If yes, then continue with block 20. If no, then continue at block 21.

Block 20: If a disengage or return to base alternative kind, set the ignore GCI vector flag *igngcv* to true.

Block 21: Otherwise set the *igngcv* flag to false.

Block 22: Set the 'more' flag to true indicating there are more alternatives to process, then return to the calling program *pkactn*.

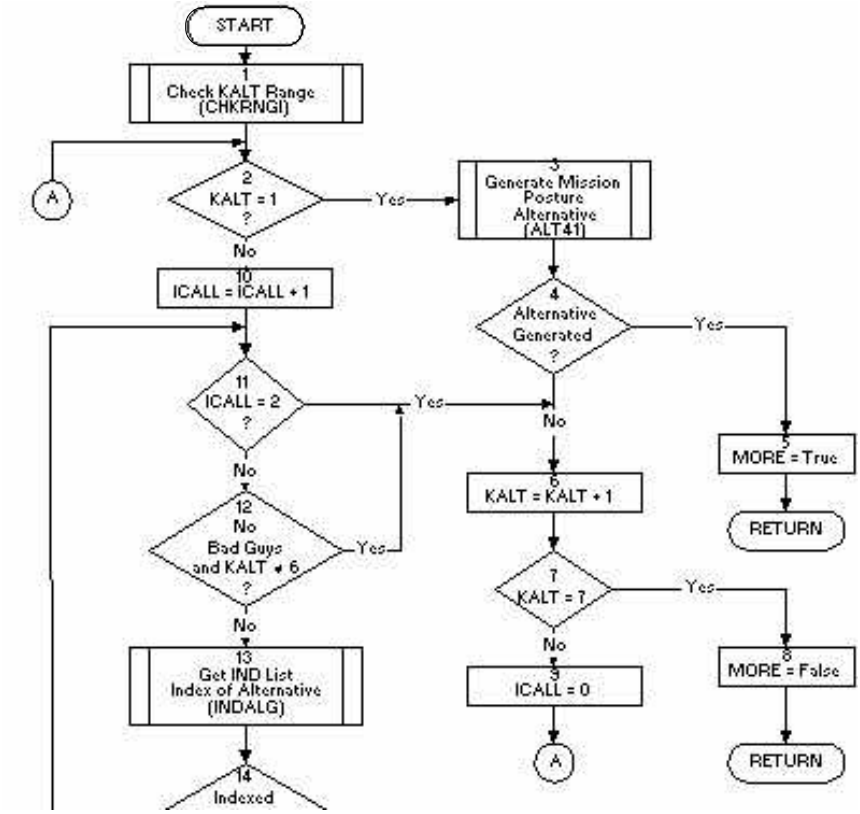

FIGURE 2.20-7. ASLCT4 Functional Flow Diagram.

Subroutine APROJ4

Subroutine *aproj4* projects the candidate alternative generated by *aslct4* in order to predict the likely result of adopting the candidate flight posture alternative. Figure 2.20-8 is the functional flow diagram which describes the logic used to implement *aproj4*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Retrieve the alternative kind *kalt* from the *althld* description array using function *int_set*.

Block 2: Ensure kind is valid (*kalt* within range) with a call to subroutine *chkrngi*.

Block 3: A computed goto on the value of *kalt* is used to select the appropriate alternative kind projection logic.

Block 4: Blocks 4 through 7 represent the mission alternative kind projection logic. Set *kill*, the weighted expected value of hostiles being lost, to 0.

Block 5: Set *risk*, the weighted expected value of friendlies being lost to 0.5**risks*, the unweighted value.

Block 6: *fmisn* is computed as a border function of *vftot* (total friendly value), *ngg* (number of good guys), and *risk*. The result of the border function is then divided by 0.9.

Block 7: *xfuel* (predicted excess fuel above bingo level) and *dteng* (predicted engagement time) are set to 0. Then *aproj4* returns to the calling routine *pkactn*.

Block 8: Blocks 8 through 12 represent the attack alternative kind projection logic. Set *wta*, the weight for a priori factors, to 0.5.

Block 9: Set *foff* and *fdef*, the weight for offensive and defensive situational factors respectively, to 1.

Block 10: Evaluate the total losses of friendlies and hostiles using subroutine *tloss*.

Block 11: *fmisn* is computed as a border function of *vftot* (total friendly value), *ngg* (number of good guys), and *risk*. The result of the border function is then weighted by 0.5.

Block 12: *xfuel* (predicted excess fuel above bingo level) and *dteng* (predicted engagement time) are set to 0. Then *aproj4* returns to the calling routine *pkactn*.

Block 13: Blocks 13 through 17 represent the evade with intent to re-engage alternative kind projection logic. Set *wta*, the weight for a priori factors, to 0.5.

Block 14: Set *foff* and *fdef*, the weight for offensive and defensive situational factors respectively, to 0.5.

Block 15: Evaluate the total losses of friendlies and hostiles using subroutine *tloss*.

Block 16: *fmisn* is computed as a border function of *vftot* (total friendly value), *ngg* (number of good guys), and *risk*. The result of the border function is then weighted by 0.5.

Block 17: *xfuel* (predicted excess fuel above bingo level) and *dteng* (predicted engagement time) are set to 0. Then *aproj4* returns to the calling routine *pkactn*.

Block 18: Blocks 18 through 22 represent the disengage alternative kind projection logic. Set *wta*, the weight for a priori factors, to 0.

Block 19: Set *foff* and *fdef*, the weight for offensive and defensive situational factors respectively, to 0.3.

Block 20: Evaluate the total losses of friendlies and hostiles using subroutine *tloss*.

Block 21: *fmisn* is computed as a border function of *vftot* (total friendly value), *ngg* (number of good guys), and *risk*. The result of the border function is then weighted by 0.5.

Block 22: *xfuel* (predicted excess fuel above bingo level) and *dteng* (predicted engagement time) are set to 0. Then *aproj4* returns to the calling routine *pkactn*.

Block 23: Blocks 23 through 27 represent the close from long range alternative kind projection logic. Set *wta*, the weight for a priori factors, to 0.75.

Block 24: Set *foff* and *fdef*, the weight for offensive and defensive situational factors respectively, to 1.0.

Block 25: Evaluate the total losses of friendlies and hostiles using subroutine *tloss*.

Block 26: *fmisn* is computed as a border function of *vftot* (total friendly value), *ngg* (number of good guys), and *risk*. The result of the border function is then weighted by 0.75.

Block 27: *xfuel* (predicted excess fuel above bingo level) and *dteng* (predicted engagement time) are set to 0. Then *aproj4* returns to the calling routine *pkactn*.

Block 28: Blocks 28 through 29 represent the return to base alternative kind projection logic. Set risk to 10**vftot*.

Block 29: Set *kill*, *fmisn*, *xfuel*, and *dteng* all to 0. Then *aproj4* returns to the calling routine *pkactn*.

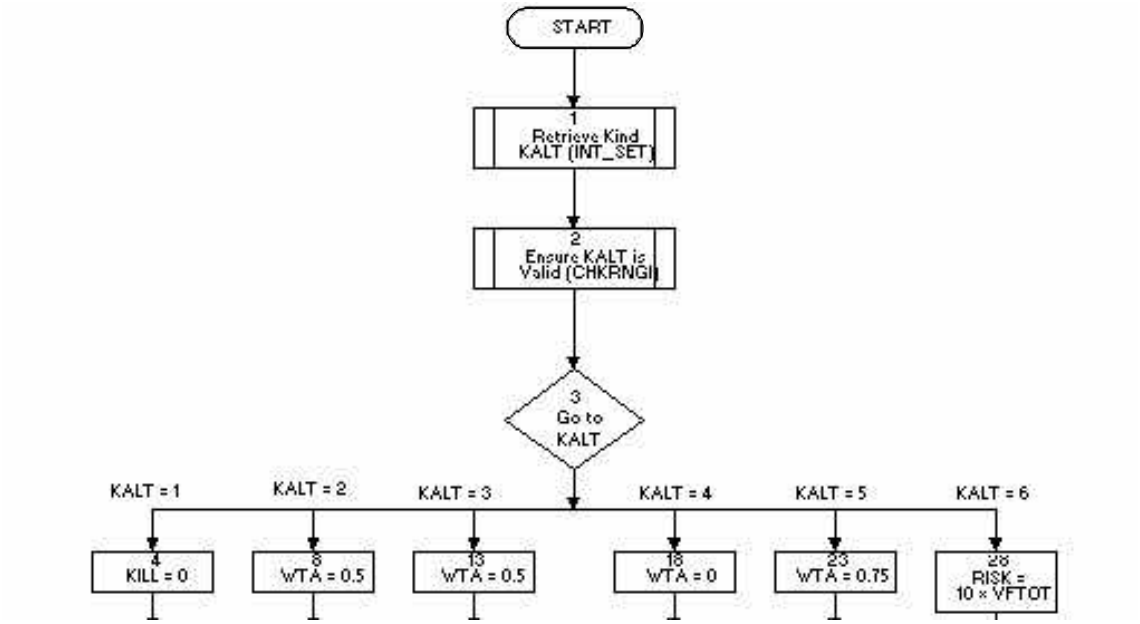

FIGURE 2.20-8. APROJ4 Functional Flow Diagram.

Subroutine AEVAL4

Subroutine *aeval4* evaluates the candidate flight posture alternative. Figure 2.20-9 is the functional flow diagram which describes the logic used to implement *aeval4*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Retrieve the alternative kind *kalt* from the *althld* description array using function *int_set*.

Block 2: Ensure kind is valid (*kalt* within range) with a call to subroutine *chkrngi*.

Block 3: Compute *altvlx*, an initial value for the candidate alternative. *fmisn* (ability to complete mission), *xfuel* (predicted excess fuel above bingo level), and *dteng* (predicted engagement time) are multiplied by their respective importance multipliers and summed.

Block 4: The net kill value *netkil* is computed as the flight leader's aggressiveness factor *aggfac* times the expected friendly loss value minus the expected hostile loss value.

Block 5: A computed goto on the value of *kalt* is used to select the appropriate alternative kind projection logic.

Block 6: Blocks 6 through 14 are performed if the alternative kind is perform mission or return to base. *altvlx* is computed as the sum of itself, *netkil*, and *rngbon*, the range bonus. These are range-independent postures and receive the full range bonus.

Block 7: Invoke subroutine *indupk* to unpack the candidate alternative descriptor *altdsc*.

Block 8: Test if a GCI drag tactic has been directed. If yes, continue at block 9. If no, continue at block 11.

Block 9: Test if the candidate alternative is GCI mission posture. If yes, then continue at block 10. If no, then branch to block 19.

Block 10: Augment the current value of *altvlx* with the addition of *gcnetk*, the GCI vectoring value.

Block 11: TestDecision on if there are any known hostile aircraft. If no, branch to block 19. If yes, continue at block 12.

Block 12: TestDecision on if the candidate alternative is a GCI mission. If no, branch to block 19. If yes, continue at block 13.

Block 13: Compute the GCI importance factor *gcifac* as a border function of message interval *gdtvmg* and amount of time since the last GCI vectoring message received (time - tmls_gci),

Block 14: Augment the current value of *altvlx* with the addition of the GCI value *gcifac***gcnetk*.

Block 15: This block is executed for alternative kinds attack immediate, evade/reengage, and disengage. Augment the current value of *altvlx* with the addition of *netkil* and *rngbon* (the range bonus) times 1-*rngfac*. These are close-in postures which get rewarded with a larger range bonus for smaller values of *rngfac*, which approaches zero as the range decreases.

Block 16: Blocks 16 through 18 are executed for the BVR attack alternative kind. The weighted range bonus is small since *rngfac* is large for long ranges. Test if a GCI drag tactic alternative is specified. If yes, execute block 17, otherwise execute block 18.

Block 17: If a GCI drag tactic augment *altvlx* with the addition of *netkil* and the weighted range bonus *rngbon***rngfac*.

Block 18: If not a GCI drag tactic augment *altvlx* with the addition of *gcnetk* (GCI vectoring value), *netkil*, and the weighted range bonus *rngbon***rngfac*.

Block 19: Test if the candidate alternative is the currently active alternative posture. If yes, continue at block 20. Otherwise, return control to the calling routine *pkactn*.

Block 20: Apply hysteresis to the current candidate. Test if *altvlx* is above 0. If yes, compute *altvlx* as in block 22, otherwise as is block 21.

Block 21: Multiply the current value of *altvlx* by the hysteresis factor (1+*hyst*) where *hyst* = 0.2. This is the final value of *altvlx*. Then return to the calling routine *pkactn*.

Block 22: Divide the current value of *altvlx* by the hysteresis factor (1+*hyst*) where *hyst* = 0.2. This is the final value of *altvlx*. Then return to the calling routine *pkactn*.

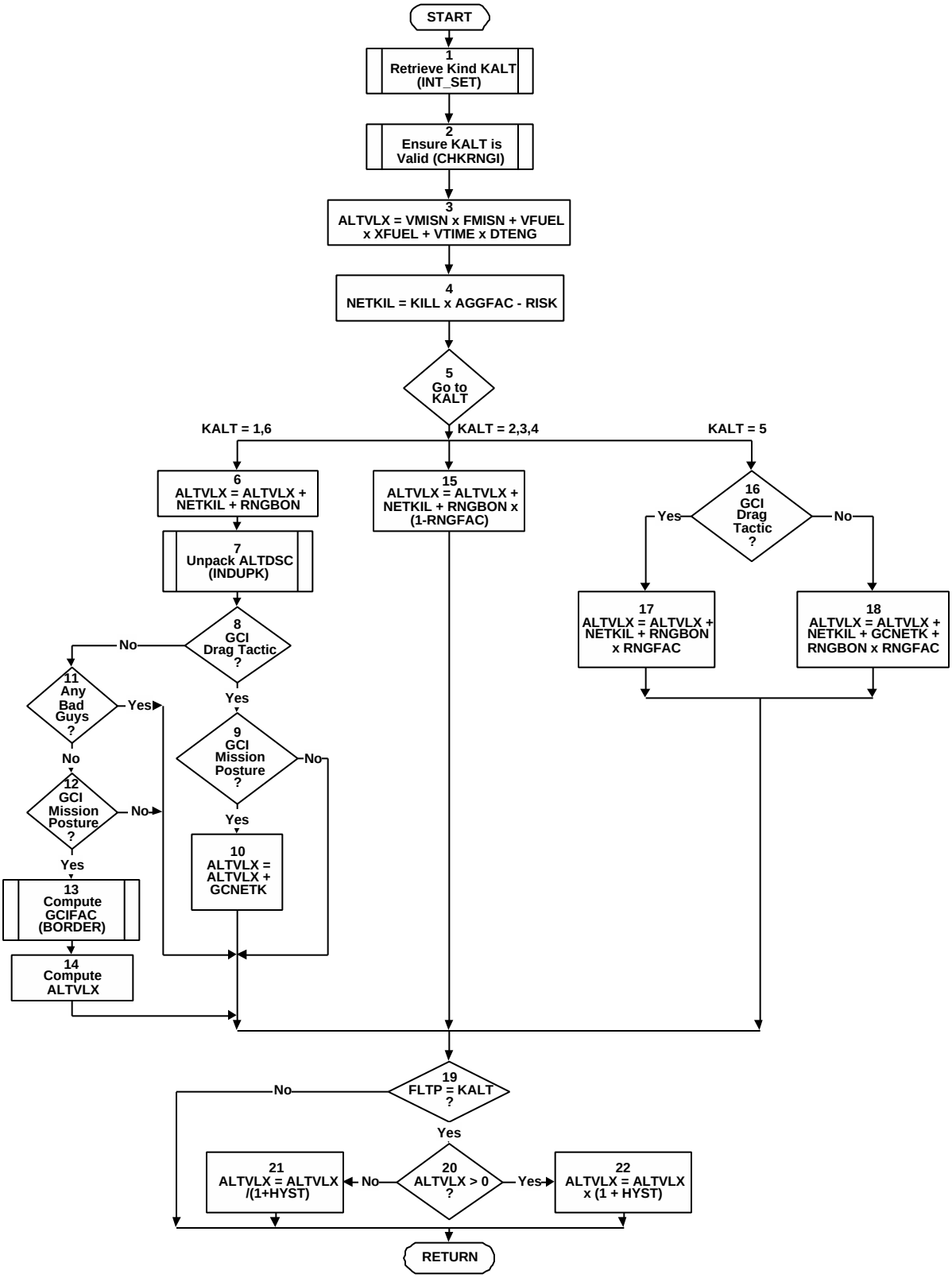

FIGURE 2.20-9. AEVAL4 Functional Flow Diagram.

Subroutine AKSHN4

Subroutine *akshn4* implements the chosen flight posture alternative by setting the mission, offensive, and defensive multipliers. Figure 2.20-10 is the functional flow diagram which describes the logic used to implement *akshn4*. The blocks are numbered for ease of reference in the following discussion.

Block 1: The alternative kind is retrieved into *nfltp* from array *althld* with utility function *int_set*.

Block 2: Ensure kind *nfltp* is valid (in range) with utility subroutine *chkrngi*.

Block 3: The value of *nfltp* is used in a computed goto to select the designated logic for setting the multipliers.

Block 4: Blocks 4 through 8 set the multipliers for the mission posture alternative kind. Block 4 unpacks alternative descriptor *altd4* with utility subroutine *indupk*.

Block 5: Ensure kind subtype *icall* is valid (in range) with utility subroutine *chkrngi*.

Block 6: *icall* is used in a computed goto to select the logic for setting the multipliers. If *icall* is 1 or 3 perform block 7. If *icall* is 2 perform block 8.

Block 7: Set the mission, offensive, and defensive multipliers to 1, 0.5***aggfac* (aggressiveness factor), and 0.5 respectively.

Block 8: Set the mission, offensive, and defensive multipliers to 1, *aggfac* (aggressiveness factor), and 1 respectively.

Block 9: For the attack alternative, set the mission, offensive, and defensive multipliers to 0.1, *aggfac* (aggressiveness factor), and min[1,*fratio*] (*fratio* = force ratio) respectively.

Block 10: For the evade and re-attack alternative, set the mission, offensive, and defensive multipliers to 0.1, max[0.5***aggfac*,0.75], and 1 respectively.

Block 11: For the disengage alternative, set the mission, offensive, and defensive multipliers to 0.5, max[0.25***aggfac*,0. 5], and 1 respectively.

Block 12: For the close from long range alternative, set the mission, offensive, and defensive multipliers to 0.1, *aggfac*, and 0.5 respectively.

Block 13: For the return to base alternative, set the mission, offensive, and defensive multipliers to 0.1, max[0.5***aggfac*,0.5], and 1 respectively.

Block 14: For the SAM site alternative (only executed for SAM sites), set the mission, offensive, and defensive multipliers to 0.1, 1, and 1 respectively.

Block 15: For the follow GCI alternative, set the mission, offensive, and defensive multipliers to 0.1, *aggfac*, and min[1.0, *fratio*], respectively.

Block 16: After setting the multipliers project the alternative with a call to subroutine *aproj4*.

Block 17: An if-elseif block is set up as a case selection on the ignore GCI vector flag *igngc4_flag*. If *igngc4_flag* is 0 branch to block 21.

Block 18: If *igngc4_flag* is 1, set the ignore GCI vector indicator *igngc4* to true.

Block 19: If *igngc4_flag* is 2 then set *igngc4* to false.

Block 20: If *igngc4_flag* is 3 or over the program is aborted with a call to subroutine *nabort*.

Block 21: Test *igngc4*. If true, perform block 20. If false, return control to the calling routine *modsel*.

Block 22: Set *tmls_gci*(*gc_vec*), the last GCI vector message time, to -1000 indicating there is no previous GCI vector message. Then return to the calling routine *modsel*.

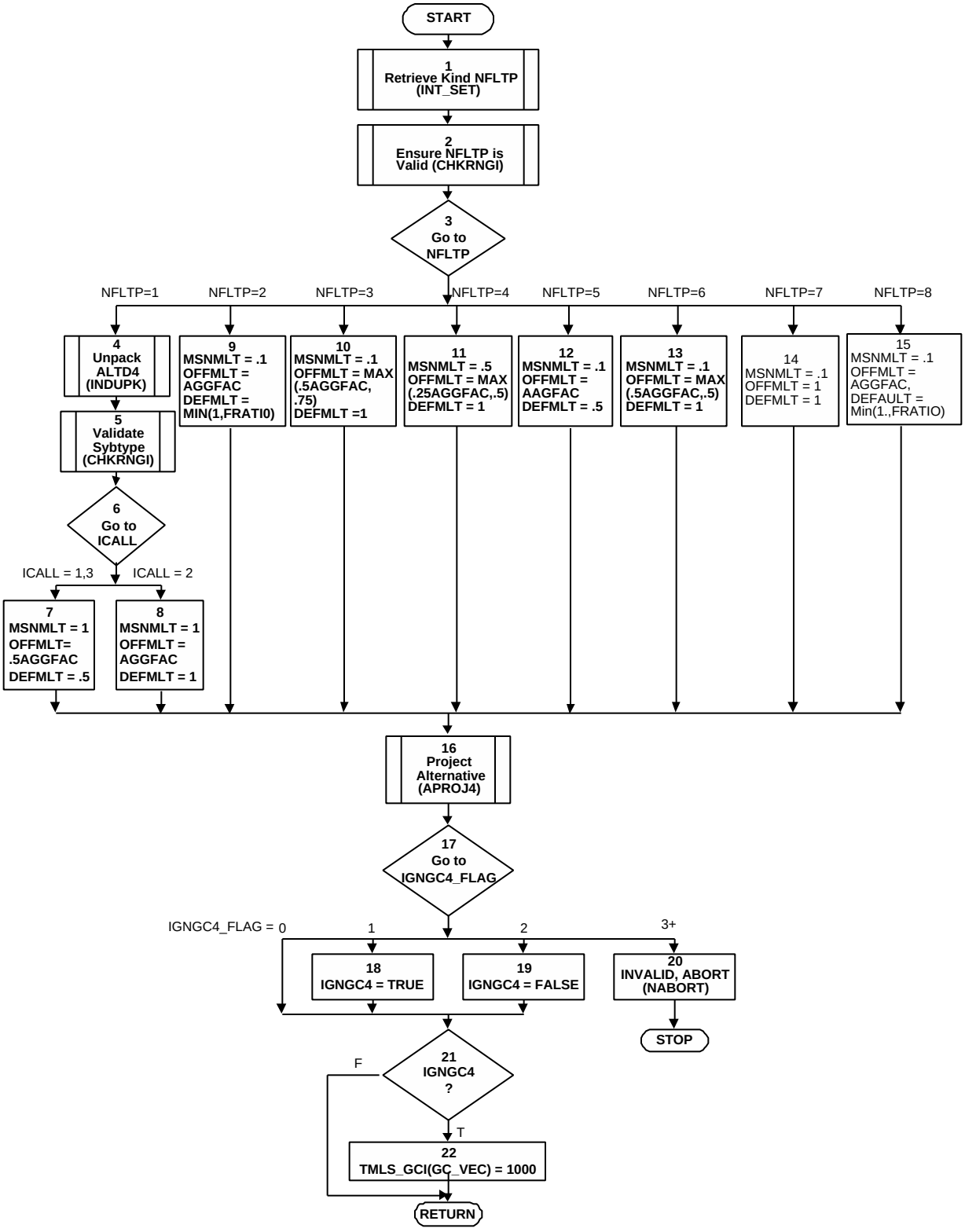

FIGURE 2.20-10. AKSHN4 Functional Flow Diagram.

Subroutine GCITAC

Subroutine *gcitac* changes the flight posture decision for GCI orders. Figure 2.20-11 is the functional flow diagram which describes the logic used to implement *gcitac*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Test if the ignore GCI message indicator *igngc4* is true or false. If true, exit and return control the calling routine *modsel*. If false, then continue at block 2.

Block 2: Test if the GCI tactic is 'follow GCI'. If not, then exit and return control the calling routine *modsel*. If it is, then continue at block 3.

Block 3: Save the current posture and tactic alternatives (*altd4* and *altd1*) into local variables.

Block 4: Begin constructing a GCI posture alternative descriptor by setting the format indicator *iactn* to 8 (flight posture) and storage size *lenalt* to *lactn*(*iactn*).

Block 5: Store *iactn* and *lenalt* into alternative description array *cactn* by using utility function *rload*.

Block 6: Unpack the current flight posture alternative descriptor *altd4*.

Block 7: Test if any there are any known hostile aircraft. If yes, perform blocks 11 through 13. If no, then perform blocks 8 through 10.

Block 8: Pack the GCI_MISSION alternative into posture alternative descriptor *altd4* using function *indpk*.

Block 9: Find the alternative index *ind4n* using function *indalg*.

Block 10: Set new flight posture kind indicator *nfltp* to 1 (mission posture).

Block 11: Pack the FOLLOW_GCI alternative into posture alternative descriptor *altd4* using function *indpk*.

Block 12: Find the alternative index *ind4n* using function *indalg*.

Block 13: Set new flight posture kind indicator *nfltp* to 8 (follow GCI posture).

Block 14: Store 0 into *cactn*(4,4) and *fltp* (current flight posture alternative) into *cactn*(5,4) by using utility function *rload*.

Block 15: Begin constructing a FOLLOW_GCI tactic alternative descriptor by setting the format indicator *iactn* to 5 (flight tactics) and storage size *lenalt* to *lactn*(*iactn*).

Block 16: Store *iactn* and *lenalt* into *cactn* by using utility function *rload*.

Block 17: Pack the FOLLOW_GCI alternative into flight tactic alternative descriptor *altd1* using function *indpk*.

Block 18: Find the follow GCI tactic alternative index *indln* using function *indalg*. Then return to the calling routine *modsel*.

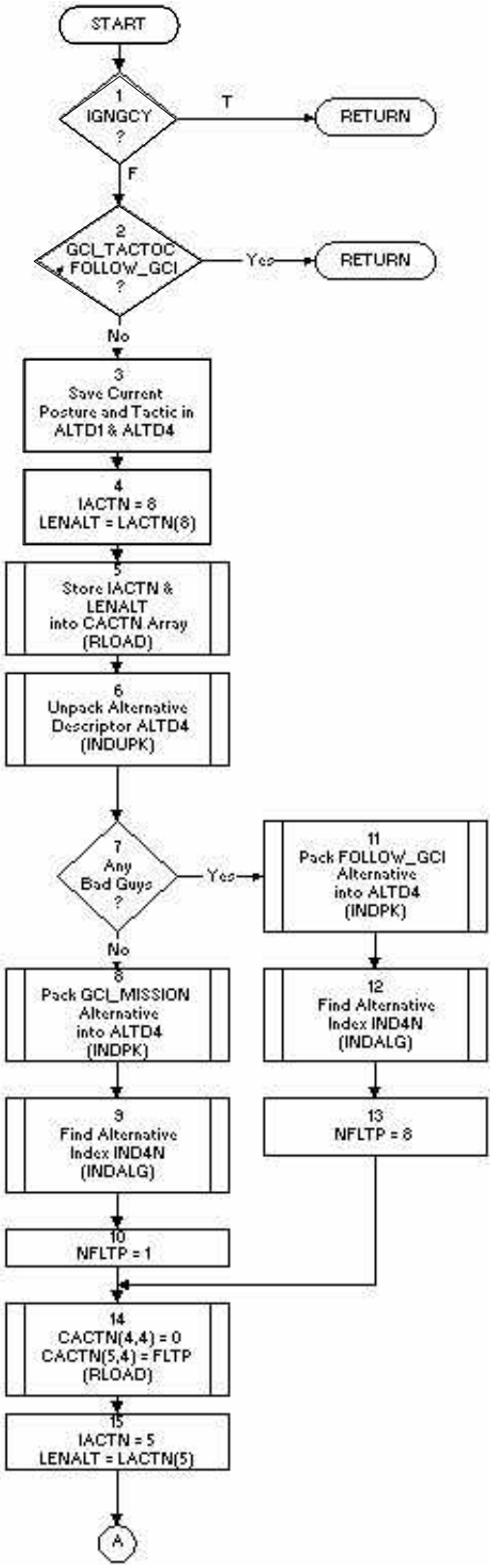

FIGURE 2.20-11. GCITAC Functional Flow Diagram (Page 1 of 2).

FIGURE 2.20-11. GCITAC Functional Flow Diagram (Page 2 of 2).

Subroutine ALT41

Subroutine *alt41* generates flight posture mission posture alternatives. Figure 2.20-12 is the functional flow diagram which describes the logic used to implement *alt41*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Progress to the next flight posture alternative subtype by incrementing *icall*.

Block 2: Test if all subtypes have been processed (if *icall* = 4). If yes, proceed to block 3. If no, proceed to block 4.

Block 3: Set subroutine parameter *submor* to false which tells the calling routine (*aslct4*) that there are no more alternatives to process. Then return to the calling routine *aslct4*.

Block 4: Find alternative index *ind* using function *indalg*.

Block 5: Test if alternative *ind* exists. If no, then go back and start this process over (block 1). If yes, proceed to block 6.

Block 6: Use *icall* in a computed goto to select the alternative mission type to process.

Block 7: If *icall* = 1 (standard mission) then blocks 7 through 11 are performed to load the alternative. This block begins the process by setting *iactn* (alternative format) to 8 (flight posture) and the storage length *lenalt* to *lactn*(*iactn*).

Block 8: Pack the flight posture alternative of subtype *icall* into posture alternative descriptor *altdsc* using function *indpk*.

Block 9: Set *althld*, the alternative logical descriptor array, element 4 with 0, element 5 with *fltp* (current flight posture), and element 6 with *kalt* (alternative kind) using utility function *rload*.

Block 10: Set the ignore GCI vector message flag *igngcv* to false.

Block 11: Set subroutine parameter *submor* to true which tells the calling routine (*aslct4*) that there are more alternatives to process. Then return to the calling routine *aslct4*.

Block 12: Blocks 12 through 14 are performed if *icall* = 2 (bomber escort mission). This block tests if the mission is not equal to 2, where 2 indicates an escort mission. If yes (not escort), then go back to the beginning (block 1) to process the next subtype. If no (is an escort mission), proceed to block 13.

Block 13: Test if no flight of bombers exists (*bflt* = 0). If yes (no bomber flight), then go back to the beginning (block 1) to process the next subtype. If no (bomber flight exists), proceed to block 14.

Block 14: Test if bomber flight *bflt* exists on the flight list *ifltx*. If no, then go back to the beginning (block 1) to process the next subtype. If yes, proceed to block 7 to generate a standard mission alternative.

Block 15: Blocks 15 and 16 are performed if *icall* = 3 (GCI mission posture). This block tests if a previous GCI vector message exists. If not proceed to block 15. If yes then proceed to block 7 to generate a standard mission alternative.

Block 16: Test if a previous GCI tactic message exists. If no, then go back to the beginning (block 1) to process the next subtype. If yes, proceed to block 7 to generate a standard mission alternative.

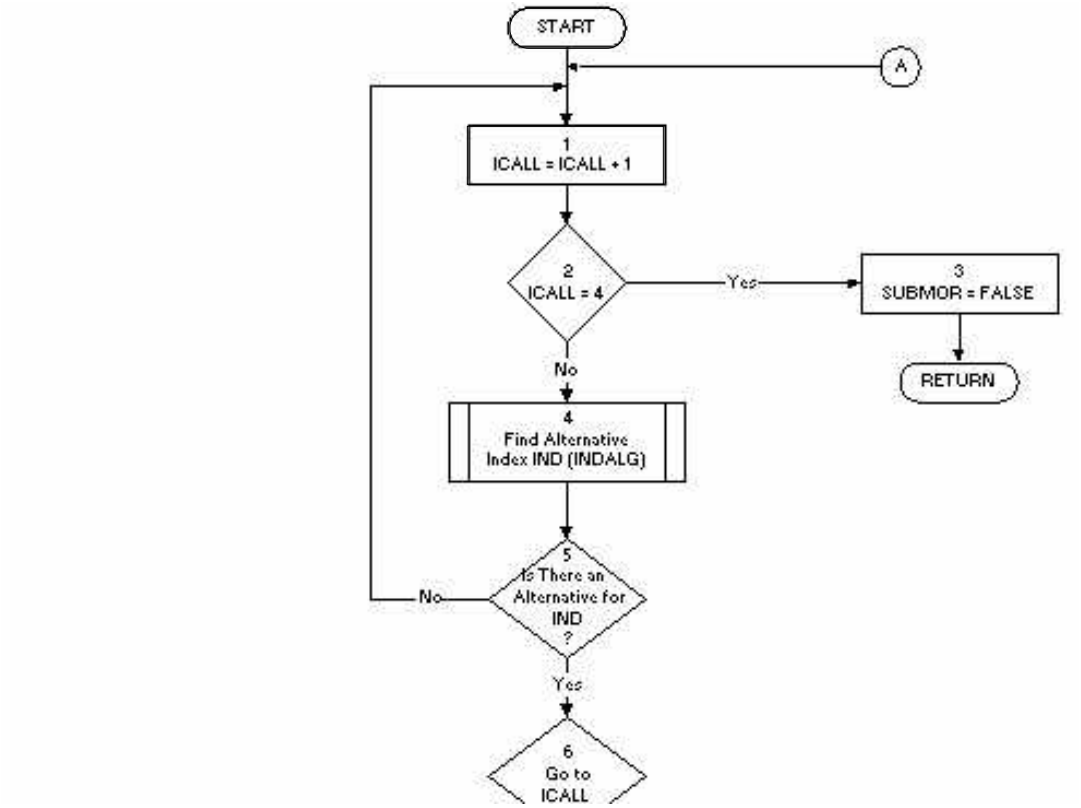

FIGURE 2.20-12. ALT41 Functional Flow Diagram.

Subroutine TLOSS

Subroutine *tloss* computes the expected hostile value killed (*kill*) and the expected friendly value lost (*risk*) for a candidate flight posture alternative. Figure 2.20-13 is the functional flow diagram which describes the logic used to implement *tloss*. The blocks are numbered for ease of reference in the following discussion.

Block 1: Compute interim a priori factor, *af*, as a function of *wta* (a priori factor weight), *killa* (a priori expected hostile value killed), and *riska* (a priori expected friendly value lost).

Block 2: Compute interim situational factor, *s,f* as a function of *wta*, *kills* (situational expected hostile value killed), *risks* (situational expected friendly value lost), *foff* (offensive situational factor weight), and *fdef* (defensive situational factor weight).

Block 3: Compute interim a priori factor weight, *wa*, as *af*/(*af*+*sf*).

Block 4: Compute interim a priori factor weight, *ws*, as 1-*wa*.

Block 5: Compute the total expected friendly value lost, *risk*, as *wa***riska* + *ws***risks***fdef*.

Block 6: Compute the total expected hostile value killed, *kill*, as *wa***killa* + *ws***kills***foff*. Then return to the calling routine *aproj4*.

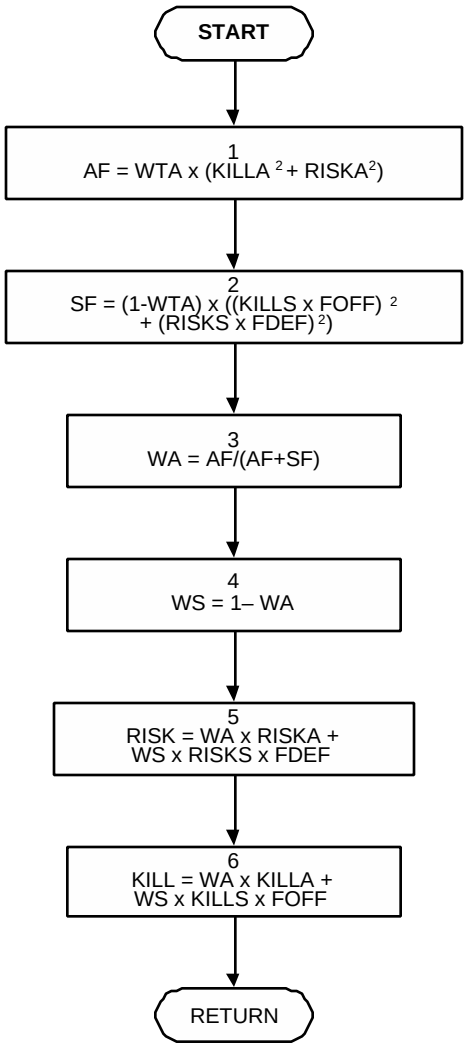

FIGURE 2.20-13. TLOSS Functional Flow Diagram.

2.20.4 Assumptions and Limitations

Entities making flight leader posture decisions are limited to aircraft. Sam sites and stand-off jammers are not considered in the decision process.

2.20.5 Known Problems or Anomalies

An error in the computed goto in Block 3 of subroutine AKSHN4 causes the mission, offensive, and defensive values for the SAM Site and Follow GCI postures to be incorrectly set to values corresponding to a mission posture instead of the desire posture. The effect of this may be to cause players in the SAM Site and Follow GCI postures to be more or less aggressive than desired.